ABACUS

# 이해하기 쉬운 코드 개발

" 이해하기 쉬운 코드는 버그가 적다. "

Hong Hyo Sang

# Contents

## I. 이해 하기 쉬운 코드

# 1. 조건문

**불필요한 코드 제거, 부정 패턴, 표현식 간소화, NullPointExecption 피하기, 대칭적 방법으로 구조화 하면 코드가 짧아지고 이해하기 쉬워 진다.**
**If문은 항상 괄호를 사용하여 블록을 쉽게 알아 볼 수 있도록 한다.**

## 01. 불필요한 비교

- 불필요한 비교는 쓸모가 없다. ( Anti-Pattern )

```
public String compare(String compareStr) {
    if (animal.isAnimal(compareStr) == true) {
        return RESULT_ANIMAL;
    } else {
        return isPlant(compareStr);
    }
}

private String isPlant(String compareStr) {
    if (animal.isPlant(compareStr) == false) {
        return RESULT_UNDEFINED;
    } else {
        return RESULT_PLANT;
    }
}
```

```
public String compare(String compareStr) {
    if (animal.isAnimal(compareStr)) {
        return RESULT_ANIMAL;
    } else {
        return isPlant(compareStr);
    }
}

private String isPlant(String compareStr) {
    if (!animal.isPlant(compareStr)) {
        return RESULT_UNDEFINED;
    } else {
        return RESULT_PLANT;
    }
}
```

## 02. 부정 패턴

- 조건문에서 부정(!)을 긍정으로 하여 이해하기 쉬운 코드를 작성 한다.

```
public String compare(String compareStr) {
    if (animal.isAnimal(compareStr)) {
        return RESULT_ANIMAL;
    } else {
        return isPlant(compareStr);
    }
}

private String isPlant(String compareStr) {
    if (animal.isPlant(compareStr)) {
        return RESULT_PLANT;
    } else {
        return RESULT_UNDEFINED;
    }
}
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/GoodCondition.java

# 1. 조건문

**불필요한 코드 제거, 부정 패턴, 표현식 간소화, NullPointExecption 피하기, 대칭적 방법으로 구조화 하면 코드가 짧아지고 이해하기 쉬워 진다. If문은 항상 괄호를 사용하여 블록을 쉽게 알아 볼 수 있도록 한다.**

## 03. Boolean 결과는 직접 반환

- boolean 을 리턴 하는 조건만은 직접 반환을 통해서 간소화 한다.
- 드 모르간 법칙을 이영해서 조건문을 부정 한다.

```java
// 지저분한 코드는 소스가 길어 지고 가독성이 떨어 진다.
public boolean isValidMessy(String name) {
    if ( name == null || name.length() <= 0 || name.trim().isEmpty()  ) {
        return false;
    } else {
        return true;
    }
}

// 지저분한 코드 제거
public boolean isValid(String name) {
    return  (name != null && !name.trim().isEmpty()) && name.length() <= 0  ;
}

// 드 모르간 법칙을 이용 적용한 간소화
// !A && !B = !(A || B ) 참
// !A || !B = !(A && B ) 참
public boolean isValidDeMogans(String name) {
    boolean isValidName = !(name == null || name.trim().isEmpty());
    return isValidName && name.length() <= 0;
}
```

## 04. 조건문이 복잡 할 경우는 조건 분리

- 조건문이 복잡 할 경우는 조건을 분리 하여 간호화 한다.
- 분리 하는 경우 처리 하는 데 있어서 중요한(빈번한) 조건을 선행 한다.

```java
// 조건문이 복잡 할 경우 분리를 통한 간소화
public boolean isValidSeparation(String name) {
    return isNameNull(name) && isNameLength(name);
}

private boolean isNameLength(String name) {
    return name.length() <= 0;
}

private boolean isNameNull(String name) {
    return !(name == null || name.trim().isEmpty());
}
```

## 05. NullPointExecption

- null 체크는 null은 우선 체크 한다,

```java
return !(name == null || name.trim().isEmpty());
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/GoodCondition.java

# 1. 조건문

**불필요한 코드 제거, 부정 패턴, 표현식 간소화, NullPointExecption 피하기, 대칭적 방법으로 구조화 하면 코드가 짧아지고 이해하기 쉬워 진다. If문은 항상 괄호를 사용하여 블록을 쉽게 알아 볼 수 있도록 한다.**

## 06. 대칭적 구조화

- 코드의 이해를 높이기 위해서 업무상 분리 할 수 있는 것은 분리 하여 개발 하는 것을 의미 한다.

```
public void 사용자권한 {

        If (알수 없는 사용자 이면 ) {
            권한 처리 ;
        } else if (일반 사용자 이면 ) {
            일반 사용자 처리;
        } else if ( 슈퍼 사용자 이면 ) {
            슈퍼 사용자 처리
        }
}
```

```
public void 사용자권한 {
        If (알수 없는 사용자 이면 ) {
            권한 처리 ;
             return;
        }


        if (일반 사용자 이면 ) {
            일반 사용자 처리;
        } else if ( 슈퍼 사용자 이면 ) {
            슈퍼 사용자 처리
        }
}
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/GoodCondition.java

# 2. 매직 넘버

**코드안의 숫자 집합은 표현상 의미가 없는 숫자는 프로그램의 동작을 제어하게 되므로 의미를 부여 하기 위해서 상수로 대체 되어야 하고 의미 있는 집합의 상수는 열거형으로 표현 하면 이해하기 쉬운 코드가 된다.**

## 01. 매직 넘버를 상수로

- 코드에 숫자로 되어 있으면 의미를 파악 하는데 시간이 걸림
- 숫자를 의미 있는 상수로 대체 하여 이해 할 수 있는 코드로 변경

```
public void setAction(int speedUnit) {
    if (speedUnit == 1) {
        setMaximumSpeed(30);
    } else if (speedUnit == 2) {
        setMaximumSpeed(60);
    } else if (speedUnit == 3) {
        setMaximumSpeed(100);
    } else {
        setMaximumSpeed(0);
    }
}
```

```
private static final int UNIT_SPEED_STOP = 0;
private static final int UNIT_SPEED_LOW = 1;
private static final int UNIT_SPEED_MIDDLE = 2;
private static final int UNIT_SPEED_MAX  = 3;

private static final int MAX_SPEED_STOP = 0;
private static final int MAX_SPEED_LOW = 30;
private static final int MAX_SPEED_MIDDLE = 60;
private static final int MAX_SPEED_MAX  = 100;

public void setAction(int speedUnit) {
    if (speedUnit == UNIT_SPEED_LOW) {
        setMaximumSpeed(MAX_SPEED_LOW);
    } else if (speedUnit == UNIT_SPEED_MIDDLE) {
        setMaximumSpeed(MAX_SPEED_MIDDLE);
    } else if (speedUnit == UNIT_SPEED_MAX) {
        setMaximumSpeed(MAX_SPEED_MAX);
    } else {
        setMaximumSpeed(MAX_SPEED_STOP);
    }
}
```

## 02. 상수를 enum 으로

- 조건문에서 부정(!)을 긍정으로 하여 이해하기 쉬운 코드를 작성 한다.

```
enum SPEED_CONTROL {
    STOP( unit: 0,  max: 0),
    LOW( unit: 1,  max: 30),
    MIDDLE( unit: 2,  max: 60),
    MAX( unit: 3,  max: 100);

    private int UNIT;
    private int MAXSPEED;

    SPEED_CONTROL(int unit, int max) {
        this.UNIT = unit;
        this.MAXSPEED = max;
    }

}

public void setAction(int speedUnit) {
    if (speedUnit == SPEED_CONTROL.LOW.UNIT) {
        setMaximumSpeed(SPEED_CONTROL.LOW.MAXSPEED);
    } else if (speedUnit == SPEED_CONTROL.MIDDLE.UNIT) {
        setMaximumSpeed(SPEED_CONTROL.MIDDLE.MAXSPEED);
    } else if (speedUnit == SPEED_CONTROL.MAX.UNIT) {
        setMaximumSpeed(SPEED_CONTROL.MAX.MAXSPEED);
    } else {
        setMaximumSpeed(SPEED_CONTROL.STOP.MAXSPEED);
    }
}
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/MagicNumber.java

# 2. 매직 넘버

**코드안의 숫자 집합은 표현상 의미가 없는 숫자는 프로그램의 동작을 제어하게 되므로 의미를 부여 하기 위해서 상수로 대체 되어야 하고 의미 있는 집합의 상수는 열거형으로 표현 하면 이해하기 쉬운 코드가 된다.**

## 03. Enum 전용

- enum 에 기능을 추가 하여 if문을 제거 하여 간결한 코드를 생성 할 수 있다.

```
enum SPEED_CONTROL {
    STOP( unit: 0,  max: 0),
    LOW( unit: 1,  max: 30),
    MIDDLE( unit: 2,  max: 60),
    MAX( unit: 3,  max: 100);

    private int UNIT;
    private int MAXSPEED;

    SPEED_CONTROL(int unit, int max) {
        this.UNIT = unit;
        this.MAXSPEED = max;
    }

    public static int getMaxSpeed(int speedUnit) {
        SPEED_CONTROL[] speed_controls = SPEED_CONTROL.values();
        SPEED_CONTROL speedcontrol =  Arrays.stream(speed_controls)
                .filter(speed_control -> speedUnit == speed_control.UNIT).findFirst()
                .orElse(SPEED_CONTROL.STOP);
        return speedcontrol.MAXSPEED;
    }
}
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/MagicNumber.java

# 3. Collection 객체 사용

**Collection 객체를 사용 하는 경우 순환 처리 하기 위해서 For문을 사용하면 인덱스의 변경을 유발 할 수 있으면 순환 처리를 객체를 변경 하여 원하지 않는 결과를 가지고 다른 방법으로 접근을 해야 함**

## 01. For-Each

- for문 사용에 대한 제약에 의해서 프로그램 구조적을 for-Each 사용
  : 인덱스의 변경에 따른 오류

```java
// i변수에 변형이 발생 하면 원하는 결과를 얻을 수 없다.
public Boolean getAnimal(String animalName) {
    for ( int i = 0; i< animals.size() ; i++) {
        // i 값에 변경이 될 수 있음
        String animal = animals.get(i);
        if (animal.equals(animalName)) {
            return true;
        }
        // i 값에 변경이 될 수 있음
        i = i + 1;
    }

    return false;
}

public Boolean getAnimalForEach(String animalName) {
    for(String animal :  animals) {
        if (animal.equals(animalName)) {
            return true;
        }
    }

    return false;
}
```

```java
public Boolean getSetAnimal(String animalName) {
    for(String animal :  animals) {
        if (animal.equals(animalName)) {
            animals.remove(animalName);
            return true;
        }
    }
    return false;
}
```

```
Exception in thread "main" java.lang.UnsupportedOperationException Create breakpoint : remove
    at java.base/java.util.Iterator.remove(Iterator.java:102)
    at java.base/java.util.AbstractCollection.remove(AbstractCollection.java:299)
    at com.hyomee.cleancode.ForCollection.getSetAnimal(UseCollection.java:60)
    at com.hyomee.cleancode.UseCollection.main(UseCollection.java:18)
```

** 순환문 내부에 pattern을 사용 하는 경우는 compile 후 사용 하면 한번 컴파일 후 사용 할 수 있음

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/MagicNumber.java

# 4. NullPointException

**NullPointException은 많이 발생 하는 오류로 여러 유형에서 발생을 하여 코드는 Null처리에 대한 방법을 제시 한다.**

## 01. 일반 적인 Null 체크 로직

- if 문을 사용 하여 처리를 하였지만 본질적으로 선택 사항이다.

```
public void generalNullCheckCode(String str) {
    if (str == null) {
        System.out.println("### Null 체크 일반 적인 개발 형태 ");
        return;
    }
    System.out.println("### 엄부 로직");
}
```

## 02. 강제 NullPointException 발생

- 사용 하는 객체가 null이면 강제로 NullPointException을 발행 하면 업무 로직 보호.

```
public void nullPointExceptionOccurrence(String  str) {
    Objects.requireNonNull(str);
    System.out.println("### :: str ::  " + str);
}
```

## 03. 다른 값으로 변환 하여 처리

- 기본 값으로 변환 하여 처리 하는 방식으로 업무 로직 보호

```
public void nullToDefaultValue(String  str) {
    str = Objects.requireNonNullElse(str,  defaultObj: "변환값");
    System.out.println("### :: 기본값 변환 :: str :: " + str);
}
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/NullProcess.java

# 5. 유연한 추상 타입

구체 타입의 코드 보다는 추상 타입의 코드가 객체의 사용에 유연성을 제공 한다.

## 01. 구체 타입 사용시 언제나 Type을 맞추어 주어야 한다.

```
ArrayList<ProductVO> productVOArrayList = new ArrayList<>();
List<ProductVO> productVOList = new ArrayList<>();
```

```
Product product = new Product();
product.setProductLikedList(productVOArrayList);
// 오류 product.setProductLikedList(productVOList);

LinkedList<ProductVO> productVOLinkedList = product.getProductList();
// 오류 ArrayList<ProductVO> productVOLinkedList1 = product.getProductList();
```

```
class Product {
    private LinkedList<ProductVO> productLikedList = new LinkedList();

    public void setProductLikedList(ArrayList<ProductVO> productArrayList) {
        productLikedList.addAll(productArrayList);
    }

    public LinkedList<ProductVO> getProductList() {
        LinkedList<ProductVO> productVOList = new LinkedList<>();
        for (ProductVO productVO : productLikedList) {
            productVOList.add(productVO);
        }
        return productVOList;
    }
}
```

구체 타입

## 02. 추상 타입은 유연성을 제공 한다.

- 사용 시점에 Type이 결정 되므로 유연성을 확보 할 수 있다.

```
ProductForAbstractType productForAbstractType = new ProductForAbstractType();
// 원하는 Type으로 파라메터 생성 해서 보내면 됨
productForAbstractType.setProducCollection(productVOArrayList);
productForAbstractType.setProducCollection(productVOList);

// 원하는 Type로 작업
LinkedList<ProductVO> productList = product.getProductList();
```

```
class ProductForAbstractType {
    private List<ProductVO> productLikedList = new LinkedList();

    public void setProducCollection(Collection<ProductVO> productVOCollection) {
        productLikedList.addAll(productVOCollection);
    }

    public List<ProductVO> getProductList() {
        LinkedList<ProductVO> productVOList = new LinkedList<>();
        for (ProductVO productVO : productLikedList) {
            productVOList.add(productVO);
        }
        return productVOList;
    }
}
```

추상 타입

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/AbstractTypeApproach.java

# 6. 변경 할 수 없는 값에 대해서는 Final

변경 하면 안되는 값에 대해서는 불변(Final)객체를 사용 하여 보호 한다. 가변 객체를 사용 하면 중간에 값이 변경 되어 원하는 값을 얻을 수 없는 경우가 발생 할 수 있다.

## 01. 가변 객체는 Data 변경이 원하는 결과가 변경될 수 있음 .

```
MutableClass seoulToDaejeon = new MutableClass ( name: "서울 to 대전 : ", value: 200);
seoulToBusan(seoulToDaejeon);
```

```
String name;
int value;
```

```
private static void seoulToBusan(MutableClass seoulToDaejeon) {

    MutableClass daejeonToBusan = new MutableClass ( name: "대전 to 부산 : ", value: 400);

    MutableClass seoulToDaejeonToBusan = seoulToDaejeon;
    // 서울 to 대전 to 부산 거리 계산
    seoulToDaejeonToBusan.add(daejeonToBusan);
    System.out.println("가변 :: 서울 to 대전 to 부산 :: seoulToDaejeonToBusan :: "
            + seoulToDaejeonToBusan.value);

    // 서울 to 대전 거리 변경를 변경 하면
    seoulToDaejeon.value = 300;
    // 서울 to 대전 to 부산 거리를 재 계산 하면
    seoulToDaejeonToBusan.add(daejeonToBusan);
    // 거리가 변경이 된다.
    System.out.println("가변 :: 서울 to 대전 to 부산 :: seoulToDaejeonToBusan :: "
            + seoulToDaejeonToBusan.value);

}
```

서울 대전 거리 변경

```
가변 :: 서울 to 대전 to 부산 :: seoulToDaejeonToBusan :: 600
가변 :: 서울 to 대전 to 부산 :: seoulToDaejeonToBusan :: 700
```

## 02. 불변 객체는 Data를 변경 할 수 없음

```
ImmutableClass seoulToDaejeonImmutable = new ImmutableClass ( name: "서울 to 대전 : ", value: 200);
finalSeoulToBusan(seoulToDaejeonImmutable);
```

```
final String name;
final int value;
```

```
private static void finalSeoulToBusan(ImmutableClass seoulToDaejeonImmutable) {

    ImmutableClass daejeonToBusanImmutable =
            new ImmutableClass ( name: "대전 to 부산 : ", value: 400);
    ImmutableClass seoulToDaejeonToBusanImmutable =
            seoulToDaejeonImmutable.add(daejeonToBusanImmutable);
    System.out.println("불변 :: 서울 to 대전 to 부산  :: "
            + seoulToDaejeonToBusanImmutable.value);

    // seoulToDaejeonImmutable.value = 300; // -> 불변 객체 이므로 수정 할 수 없음 ;
}
```

```
불변 :: 서울 to 대전 to 부산  :: 600
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/UseFinal.java

ABACUS

# 7. Null 객체 반환

**메서드가 null을 반환 하면 코드에서 NullPointException 발생 할 수 있어서 매번 명시적으로 확인 하는 데 이것을 해결 하기 위한 방법**

## 01. 일반적인 코드

- NullPointException을 피하기 위해서 검증 한다.

```
City city = ExplicitNullReturn.getCity( cityName: "서울");
System.out.println( " city.getCity() :: " +  city.getCity() );
city = ExplicitNullReturn.getCity( cityName: "부산");
if (city == null) {
    System.out.println(" 부산 ::  null ... ");
} else {
    System.out.println(" city.getCity() : " + city.getCity());
}
```

```
class ExplicitNullReturn {
    static List<City> cities = Arrays.asList(
            new City("서울"),
            new City("대전")
    );
    static City getCity(String cityName) {
        for (City city: cities) {
            if (city.getCity().equals(cityName)) {
                return city;
            }
        }
        return null;
    }
}
```

## 02. Final을 사용한 null 객체 사용

```
City city = ObjectNullNeturn.getCity( cityName: "서울");
System.out.println( " city.getCity() :: " +  city.getCity() );
city = ObjectNullNeturn.getCity( cityName: "부산");
System.out.println( " 부산 :: city.getCity() :: " +  city.getCity() );
```

```
class ObjectNullNeturn {

    private static final City city = new City("");

    static List<City> cities = Arrays.asList(
            new City("서울"),
            new City("대전")
    );

    static City getCity(String cityName) {
        for (City city: cities) {
            if (city.getCity().equals(cityName)) {
                return city;
            }
        }

        return city;
    }
}
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/UseFinal.java

# 8. Optional 사용

**메서드가 null을 반환 하면 코드에서 NullPointException 발생 할 수 있어서 매번 명시적으로 확인 하는 데 이것을 해결 하기 위한 방법**

.

## 01. Optional 사용

- Optional을 사용 하고 ifPresent를 이용해서 null이 아닌 경우 만 실행 하게 한다.

```java
class NullReturn {
    private SamleDTO samleDTO;

    public SamleDTO getSamleDTO() {
        return samleDTO;
    }
}
```

```java
class OptionalReturn {
    private SamleDTO samleDTO;

    public Optional<SamleDTO> getSamleDTO() {
        return Optional.ofNullable(samleDTO);
    }
}
```

```java
class SamleDTO {
    private String name;
    private String code;

    public void printSampleDTO(String str) {
        System.out.println("### 출력 " + str );
    }
}
```

```java
NullReturn nullReturn = new NullReturn();
SamleDTO samleDTONullPeturn= nullReturn.getSamleDTO();
if (samleDTONullPeturn == null) {
    System.out.println("### samleDTONullPeturn SamleDTO은 null 입니다.");
} else {
    samleDTONullPeturn.printSampleDTO( str: "Data....");
}
```

```java
OptionalReturn optionalReturn = new OptionalReturn();
SamleDTO samleDTOOptionalNull = optionalReturn.getSamleDTO()
        .orElse( other: null);
if (samleDTONullPeturn == null) {
    System.out.println("### samleDTOOptionalNull :: SamleDTO은 null 입니다.");
} else {
    samleDTONullPeturn.printSampleDTO( str: "Data....");
}
```

```java
optionalReturn.getSamleDTO().ifPresent(samleDTO2 -> samleDTO2.printSampleDTO( str: "Data...."));
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/ComplicatedComtion.java

# 9. Collection객체는 Stream사용

**Collection처리는 함수형 프로그램 방식이 읽기 쉽다.**

## 01. 일반 적인 Collection 객체 코드

- 무엇을 해야 하는지. 루프와 조건, 변수 할당과 같은 것을 고려 해야 한다.
- 코드를 한 줄 한 줄 따라가면서 읽고 해석 해야 한다.

```
class Books {
    List<Book> books = Arrays.asList(
            new Book( bookId: "01",  bookName: "Spring",  category: "IT"),
            new Book( bookId: "02",  bookName: "Spring",  category: "문학"),
            new Book( bookId: "03",  bookName: "JAVA",  category: "IT")
    );

    public long countCategory(String category) {
        List<Book> bookList = new ArrayList<>();

        for (Book book: books) {
            if (book.getCategory().contains(category)) {
                bookList.add(book);
            }
        }

        return bookList.size();
    }
}
```

## 02. Stream 사용

- 무엇을 해야 하는지 명시적으로 표시 한다.
- filter : Predicate
- map : Function

.

```
class StreamBooks {
    List<Book> books = Arrays.asList(
            new Book( bookId: "01",  bookName: "Spring",  category: "IT"),
            new Book( bookId: "02",  bookName: "Spring",  category: "문학"),
            new Book( bookId: "03",  bookName: "JAVA",  category: "IT")
    );

    public long countCategory(String category) {

        return books.stream().filter((book -> book.getCategory().contains(category))) Stream<Book>
                            .map(book -> book.getBookId()) Stream<String>
                            .distinct()
                            .count();
    }
}
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/CollectionToStream.java

# 10. 복잡한 조건은 메서드 참조

**Collection를 처리 할 때 복잡한 조건은 메소드 참조로 코드를 간결히 작성 한다.**

## 01. 복잡한 조건은 메소드 참조

- Stream에 복잡한 코드가 있으면 코드 반독이 쉽지 않으므로 메소드 참조를 통해서 읽기 쉬운 코드로 변환 한다.

```
public long countCategory(String category) {
    return complicatedBooks.stream().filter((complicatedBook -> complicatedBook.getCategory().contains(category)
                                    && complicatedBook.getBookName().equals("Spring"))) Stream<ComplicatedBook>
            .map(book -> book.getBookId()) Stream<String>
            .distinct()
            .count();
}
```

참조 메서드 :

```
private boolean isContains(ComplicatedBook book, String targetCategory) {
    return book.getCategory().contains(targetCategory) && book.getBookName().equals("Spring");
}
```

참조 메서드 참조로 변경

```
public long countCategoryByMethod(String category) {
    return complicatedBooks.stream().filter( complicatedBook -> isContains(complicatedBook, category)) Stream<ComplicatedBook>
            .map(ComplicatedBook :: getBookId) Stream<String>
            .distinct()
            .count();
}
```

새로운 객체로 리턴

```
public List<ComplicatedBook> getCategoryByMethod(String category) {
    return complicatedBooks.stream().filter( complicatedBook -> isContains(complicatedBook, category))
            .map(ComplicatedBook::new)
            // .map(complicatedBook -> new ComplicatedBook(complicatedBook))
            .collect(Collectors.toList());
}
```

소스 : https://github.com/hyomee/JAVABASIC/blob/main/src/com/hyomee/cleancode/ComplicatedComtion.java

# THANKS

ABACUS

www.iabacus.co.kr

Tel. 82-2-2109-5400

Fax. 82-2-6442-5409