1. 출력하기

```
Public class Main {
        Public static void main(String[] args) {
                System.out.println("안녕 자바!");
                System.out.println("변경 내역 추가");
}
}
```

2. 자바 기본 변수

```
Package variable'
Public class VariableTest {
        Public static void main(String[] args) {
                System.out.println("variable(변수)테스트);
        //변수를 만드는 방법
        // 1. 데이터 타입을 적는다 (int, float, double, long)
        //      int: 정수형(숫자-음수, 양수)
        //      long: 비슷함
        //      float: 소수점
        //      double: 조금 더 정밀한 소수점(실제 작업하면서
                                double 사용할 일은 거의 드뭄)
        //2. 변수 이름을 작성한다 (이름은 우리가 만들고 싶은
```

대로 만들 수 있다)

```
// 3. 필요하다면 초기화를 진행한다.


//결론적으로 변수를 만들 때 가장 중요한 것은
//변수 이름인데 명시성을 통해 함께 작업하는 팀원들에
        에게
//이것이 무엇을 의미하는 것인지 명확하게 전달하기 위
        목적이 가장 중요합니다.
int appleCount = 3;
int grapeCount = 5;
int totalCount = appleCount + grapeCount;
/*
int n1 = 3, n2 = 5;
int res = n1 + n2;
*/


//문자열 + 숫자의 경우엔 앞에 문자열이 나왔기 때문에
//자동으로 숫자를 문자열 처리하여 화면에 출력합니다.
System.out.println("totalCount = " + totalCount);
//final을 사용하는 이유 : 우선 상수로 고정시킬 수 있다
```

```
//는 이점이 있음
//상수로 고정시킬 수 있다는 것의 이점이란 무엇인가?

//1. 아래 있는 [비교 대상]을 보면 3.3f를 직접 기입하고
        있으므로 향후 프로그램이 커지면 직접 3.3f를 작성
        한 부분을 모두 찾아서 TAX 변경에 따라 모든
        코드를 수정해야 하는 불편함이 발생합니다.
        반면 TAX 상수에 숫자를 기입하고 이 상수를
        사용한다면 변동 상황이 발생할 때 해당하는 TAX
        수치값만 한번 변경하면 모든 작업이 일괄처리
        되므로 편합니다.
final float FULL_PERCENT = 100;
final float TAX = 3.3f;
int income = 1000000;

System.out.println("프리랜서 세전 수입: " + income +
        ", 세후: " + income * (FULL_PERCENT – TAX) /
        FULL_PERCENT);
/* [ 비교 대상]
System.out.println("프리랜서 세전 수입 : " + income +
```

```
        ",  세후: " + income * (FULL_PERCENT – 3.3f) /
        FULL_PERCENT);
        */
//final을 사용할 때 가지는 이점 두 번째
//2. 불변 객체 (Immutable Object)
//클래스를 인스턴스화 하여 객체를 만들었고 이것이
        불변이라면 무엇이 좋을까요?
// TAX = TAX + 4;
//위 코드는 TAX가 final 이기 때문에
//새로운 값을 대입하거나 덧셈, 뺄셈 등등이 불가능합
        니다.
//결론적으로 입력되는 값을 변경하지 못하게 막음으로
        서 원래 동작해야 하는 동작의 무결성을 보장하게
        됩니다.

//예 ) 1 – 예금, 2 – 출금, 3 – 조회
//회사 프로그램이 만들어져 있는 상태(예금, 출금)
//신입이 조회를 만들고 있는 상황입니다.
//만들던 중 코드에 계속, 1과 2가 날아오니까
//신입은 그냥 들어온 숫자를 3으로 대입하고 문제를
```

```
        // 해결했습니다. 이 상황에선 예금을 했더니 조회가 되고
        //출금을 했더니 조회가 되고
        //조회를 했더니 조회가 되는 끔찍한 상황이 연출됩니다.
        //이것을 원천 차단하는 방법으로서 final을 사용합니다.
        //3을 대입하는 행위 자체를 차단하는 것이죠.
}
}
```

3. if 문 테스트

```
package flowControl;


    public class IfTest {
        public static void main(String[] args) {
            final int PERMIT_AGE = 18;
            final int inputAge = 15;


        //if 문을 만드는 방법
        // 1. If를 작성하고 소괄호()를 작성하고 중괄호{}를 작성
        //2. 소괄호 내부에 조건식을 작성
        //3. 중괄호 내부에는 조건이 만족된 경우 동작할 코드를
            작성
```

```
If(PERMIT_AGE < inputAge) {
        System.out.println("입장 가능하십니다!");
} else {
        System.out.println("입장 불가능하십니다!");
}

final int PERMIT_KIDS = 13;
if(PERMIT_AGE < inputAge) {
        System.out.println("성인용입니다!");
}
if(PERMIT_KIDS< inputAge) {
        System.out.println("아동용입니다!");
}

//만약 if, else if, else if, else if 형태로 코드가 작성되면
//조건식을 첫 번째 if가 만족되지 않았을 때 else if를 보
//게 되므로 기본적으로 해당 else if에서는 if의 조건
//또한 만족하지 않음을 내포하게 됩니다.
//그리고 그 다음 else if에서는 맨 처음 if가 만족되지
//않고, 그 다음 else if를 만족하지 않고, 그리고 현재의
```

```
            //else if 조건을 만족해야 합니다.


            //그러므로 depth(깊이)가 깊어질수록 코드를 파악하기
            //위한 혼동이 가중된다는 문제가 있습니다.
            //이와 같은 이유 때문에 코드를 작성할 때 if, else if,
            //else if보다는 그냥 if, if, if가 더 좋습니다.


            /* [비교 대상]
            If(PERMIT_AGE < inputAge) {
                    System.out.println("성인용입니다!");
        } else if (PERMIT_KIDS < inputAge) {
                    System.out.println("아동용입니다!);
        }
            */
    }
}
```

4. while 문 테스트

```
package flowControl;
```

```
public class WhileTest {
    public static void main(String[] args) {
            int idx = 0;
            final char ch = 'A';
            //while 문 작성 방법
            //1. 일단 while을 적고 소괄호()를 작성하고 중괄호 {}를
                작성합니다.
            //2. 소괄호 내부에 조건식을 작성합니다.
            //   <<< --- idx < 10의 엄밀한 뜻을 아래와 같습니다.
            //idx 변수의 값이 숫자 10 보다 작은게 맞니 ? 라고
            //물어보는 것입니다.
            //그리고 그 답으로 참(True) 혹은 거짓(False)이 튀어
            //나옵니다.
            //while(true) 혹은 while(false)로 치환되므로
            //조건이 만족되면 루프를 돌고 만족되지 않으면 루프를
            //빠져나오게 됩니다.
            //3. 중괄호 내부에 조건이 만족되는 동안 반복시킬 코드
            //를 작성합니다
            while (idx < 10) {
                System.out.println("idx: " + idx + ", 안녕 : " + (char)
```

```
                    (Ch + idx));
            //char 타입의 변수들은 독특한 특성을 가지고 있습니다.
            // ASCII 코드 특성인데 실제 알파벳 'A'는 숫자 65에 해당
            //합니다. 이와 같은 이유로 실질적으로 숫자값들과 덧셈,
            // 뺄셈등의 연산을 수행 할 수 있습니다.

            // 위 규칙에 따르면 'B'는 66이라는 것도 알 수 있습니다.
            idx++;
        }
    }
}
```

5. switch test

```
package flowControl;

import java.util.Scanner;

public class SwitchTest {
    public static void main(String[]  args) {
            //Scanner는 키보드 입력 처리를 위해 사용하는 객체
```

```
//여러분이 실제 서비스를 개발하면서 아래 코드를 사용
//할 일은 없지만 현재 콘솔 상황에서 사용자 입력을
//받기 위해 아래 코드가 사용된다 보면 되겠습니다.
//사용자 입력이란 구체적으로 키보드 입력을 의미합니다.
//System.in 이란 입력시스템을 의미하므로
//입력 장치에 해당하는 키보드를 의미한다 봐도 무방

//결론: 사람의 키보드 입력을 받고 싶으면 아래 코드
            한줄을 입력하세요.
Scanner scan = new Scanner(System.in);

// boolean: 참/거짓을 표현하는 자료형입니다.
Boolean isLoop = true;

//isLoop가 true인 동안 반복
While (isLoop) {
        System.out.println("숫자를 입력하세요:");
//키보드 입력으로 int 타입을 수신한다면 nextInt()를
//사용합니다. 만약 double 타입을 원한다면
//nextDouble() , float 타입을 원하면 nextFloat() 형태로
```

```
//사용하면 됩니다.
Int inputNumber = scan.nextInt();

//switch 문을 작성하는 방법
//1. switch를 적고 소괄호()를 작성하고 중괄호{}를 작성
//2. 소괄호 내부에 switch case에서 사용할 조건을 적기
// -> 현재 케이스에서 inputNumber는 숫자이므로
//case 0: 의 의미는 '입력된 숫자가 0이면' 이라는 뜻을
//가집니다.
//case 1:은 ' 입력된 숫자가 1이면' 이란 뜻입니다.
//3. 중괄호 내부에는 case 조건들을 적고
//각 조건에 대응하는 코드를 작성하면 됩니다.
switch(inputNumber) {
        case 0:
                //숫자 0이 들어오는 경우 isLoop를 false
                        (거짓)으로 바꿈
                System.out.println("종료");
                isLoop = false;
                break;
        case 1:
```

```
                System.out.println("입금!");
                break;

            Case 2:
                System.out.println("출금!");
                break;

            Case 3:
                System.out.println("조회!");
                break;

            default:
                System.out.println("그런 명령은 존재하지
                                    않습니다!");
                break;
            }
        }
    }
}
```

6. for 테스트

```
package flowControl;


public class ForTest {
    public static void main(String[ ] args) {
            final int START = 3;
            final int END = 10;


            int index = START;


            //for문을 만드는 방법
            //1. for을 작성하고 소괄호()를 작성후 중괄호{}를 작성
            //2. 소괄호 내부는 아래와 같이 구성됩니다.
            //      (초기화;조건;증감)
            //      - 여기서 초기화란 for문을 최초로 만나는 순간
            //에만 동작하게 됩니다. 그러므로 없어도 됩니다.
            //      - 조건은 while, if, switch 등에서 봤던 조건식과
            //동일합니다. 조건을 만족하는 동안 for문이 사용됩니다.
            //      - 증감의 경우에도 없어도 됩니다.
            //표현을 조금 더 예쁘게 만들어주기 위해 증감파트가
```

```
//존재한다 봐도 무방합니다.
//3. 중괄호 내에는 for문을 반복하며 작업할 내용을
      적어줍니다.

//중요한 것은 어찌되었든 for문은
//조건 파트가 참인 동안은 언제든지 반복된다는 것
// 초기화나 증감파트는 곁다리
for(; index < END; ) {
        System.out.println("index = " + index+++);
  }

   /*
  for (int index = START; index < END; index++) {
        System.out.println("index = " + index);
}
*/

//루프를 돌면서 덧셈을 하려면 반드시 아래와 같이
//외부에 0으로 초기화된 변수를 가지고 누산해야 합니다.
int sum = 0;
```

```
        int count = 0;

        for (int idx = START; idx < = END; idx++) {
            // sum += idx;
            // sum은 sum + idx라는데 왜 값이 누산?
            // '=' 연산자는 '~와 같다가 아닙니다'
            // 오른쪽에 있는 정보를 왼쪽에 대입
            // '=' 연산자는 대입 연산자
            sum = sum + idx;
            System.out.println("count = " + (++count) + ", sum
                                    = " + sum);
}
            System.out.println("3~10까지의 합: " + sum);

            // 조건이 없으므로 무조건입니다.
            // for(,,) {
            // System.out.println("무한 반복");
            //   }

            for (int i = START; i < = END; i++) {
```

```
                    // '%' 연산자는 나머지 연산자입니다.
                    // (mod 2)와 동의어이며 이건 몰라도 됩니다.

                    // 현재 i 값을 2로 나눈 나머지가 0이냐를 묻는 것
                    // 2로 나눈 나머지가 0이란 것은 짝수임을 의미
                    // 고로 이 로직은 3 ~ 10사이의 홀수만 출력
                if( i % 2 == 0 ) { continue; } // continue는 skip과 동일

                System.out.println("i = " + i);
                }
        }
}
```

7. 랜덤 숫자 만들기

```
Package math;

Public class RandomTest {
        Public static void main(String[] args) {
            final int START = 0;
            final int END = 100;
```

```
 final int TEST_MAX = 77;
 final int MAX = 15;
 final int MIN = 5;
 //랜덤을 만드는 방법
//1. Math.random()을 작성합니다.
//2. 최소값과 최대값을 확인합니다.
//3. 최소값은 더하기로 표기해주세요 (아래 코드에선 + 1)
//4. 최대값은 곱하기로 표기하고
//    실제 최대값 계산은 곱하기하는 값 + 최소값 – 1
//    (아래에선 100 + 1 – 1로 100에 해당함)
//    그러므로 1 ~ 100까지의 숫자를 만듭니다.
final int randomNumber = (int)(Math.random() * 100) +1;

System.out.println("randomNumber = " + randomNumber);

int randomValue = 0;
for (int i = START; i < END; i++) {
      randomValue = (int)(Math.random() * (MAX – MIN + 1))
                              +MIN; // 5 ~ 15
      System.out.println("I = " + i + ", randomValue = " +
```

```
                                        randomValue);
                }
        }
}
```

8. Stack에 배열 만들기

```
Package array;


Public class StackArrayTest {
        Public static void main(String[] args) {
                final int START = 0;


                //배열을 만드는 방법 (일단 final은 제끼세요)
                //1. 데이터 타입을 적고 대괄호[]를 적습니다.
                //2. 변수 선언하듯 변수 이름을 작성합니다.
                 //   당연히 이름이기 때문에 전달력 및 표현력 중요
                //3. 필요다하면 중괄호를 열고 초기화를 하거나
                //     또는 new 데이터타입(1번) [] 형태로
                //     Heap에 메모리 할당을 강제할 수 있습니다.
                final int[] numberArray = { 1, 2, 3, 4, 5};
```

```
  //여기서 이야기 한 Heap과 대조되는 것이 Stack입니다.
  //현재 위 코드는 Stack이라는 지역 변수에 설정하는
  //배열이며 new를 통해 할당하는 경우엔 Heap이라는
  //공간에 할당합니다. 우리는 로우 시스템 개발자가
  //아니므로 여기서 더 깊게 들어갈 필욘 없습니다.
  // 그냥 아 ! new를 했기 때문에 Heap에 있구나 정도

  //그렇다면 궁극적으로 Stack과 Heap의 차이가 무엇인가?
  //Stack은 앞서서 주사위 문제에서도 봤듯이
  //Loop { final int data } 형태가 있다면
  //data 변수가 루프마다 초기화 되는 것을 볼 수 있다.
  //이런 지역변수 특성을 가지는 녀석들은 죄다 Stack

  //반면 new를 해서 Heap에 설정되는 정보들은
  //메모리에 상주하게 됩니다. 그러므로 언제 어디서든
 //데이터에 접근 할 수 있게 됩니다. 자바 개발자에게
 //있어 둘의 차이점이라면 현재 이 내용이 가장 크다
//볼 수 있겠습니다.

 //결론: Stack은 중괄호 {} 내에서 사용됨
```

```
//  Heap은 new 하고 이후로 사용됨


//배열의 경우 아래와 같이 numberArray.length 형태로
// 전체 길이를 파악 할 수 있습니다.
for(int i = START; I < numberArray.length; i++) {
  //배열이라는 녀석은 메모리 공간상에 순차적으로
  //배치, 메모리 공간이라는 것이 생소하다면
  //아래와 같이 박스들이 일렬로 나열되어 있는 상황을
  //생각해도 됩니다.

  // 0    1     2    3   4   <<-- 5개 있지만 0 ~ 4로 표현됨
  // [    ][    ][    ][    ][    ]

// 주의할 부분이라면 배열의 시작이 0부터라는 것에 주
   의를 해주세요.
  // 우리는 1부터 100까지라고 말하지만
  // 배열은 0부터 99까지 움직이게 됩니다.

  // final int[] numberArray = { 1, 2, 3, 4, 5 };
  // 위쪽에 이 코드는 아래와 같이 배치되어 있습니다.
```

```
            //    0      1      2      3      4
            // [ 1 ][ 2 ][ 3 ][ 4 ][ 5 ]

            // 그러므로 numberArray[0]은 숫자 1을 표현하고
            // numberArray[1]은 숫자 2를 표현하며
            // numberArray[2]는 숫자 3
            // numberArray[3]는 숫자 4
            // numberArray[4]는 숫자 5

            // 위 형태로 동작하여 for 루프에서
            // i값이 증가함에 따라 모든 배열의 원소들을 출력하게 됩니다.
            System.out.println("배열 출력: " + numberArray[i]);
        }

        System.out.println();

        //for의 변형 버전 foreach라고도 이야기함
            //foreach 사용법
            //1. 배열의 데이터 타입을 작성합니다 (여기선 int)
```

```
//2. 배열의 원소를 표현할 이름을 적당히 지정합니다 (여기선 num)
    //3. 콜론 하나 찍습니다 (:)
//4. 정보를 하나씩 꺼내 올 배열을 적어줍니다 (여기선 numberArray)
    for (int num: numberArray) {
        System.out.println("배열 원소 출력: " + num);
    }

//비즈니스가 확장됨에 따라 서비스가 커졌고
//그에 따라 데이터의 규모도 커졌다 (여기선 배열에 정보가 많다고 가정)
//이런 상황에서 for(초기화; 조건; 증감) 과 foreach가 있다면 누가 더 좋을까 ?

//첫 번째 - 이유 -> 초기화나 조건등이 있어 향후 변경에 유리하다 생각됨
//두 번째 - 이유 -> 전자는 계속 조건식을 검사해야함
//이런 이유 때문에 사실 데이터를 요청하면
//뭉태기로 한 번에 쏘지 않고 필요한 정보만 선별해서 쏘게 됩니다.
```

```
            //가령 AWS에서 1000만명 분량의 데이터를
            //검색 한 번에 다 가져오게 되면 요금 폭탄맞고 회사 망합니다.
            //그래서 서비스 관점에선 이런 조각 내기 개념으로서 페이징(Paging)이라는 것을 합니다.
            //OS 레벨에서 이야기하는 Paging과는 다른 개념이니 주의합시다!
            //(여기서 이야기하는 Paging은 데이터 조각내기 관점이라 보면 됩니다)
        }
    }
```

9. 메모리 변경 금지

```
package array;

public class HeapStackTest {
    public static void main(String[] args) {
        final int START = 0;
        final int ALLOC_ARRAY_NUMBER = 5;

        //Heap에 할당된 메모리 변경 금지를 요청한 것이고
        //내부에 배치하는 것에는 영향을 받지 않기 때문임.
        final int[] numberArray = new int[ALLOC_ARRAY_NUMBER];
```

```
//아래와 같이 새로운 메모리를 할당해서 전달하는 것을 막고 있습니다.
//조금 풀어보자면 객체를 상수화 하느냐
//객체 내부의 값을 상수화 하느냐의 관점으로 봐야합니다.
//현재 관점은 객체를 상수화하였기 때문에 다른 객체 대입이 막히는 모습입니다.
// numberArray = new int[8];

for (int i = START; i < ALLOC_ARRAY_NUMBER; i++) {
 numberArray[i] = i + 1;
 //printf의 경우 format을 출력한다는 뜻으로 printf 입니다.
 //format은 %d의 경우 정수형(int)
 //%s의 경우(String)인데 필요 없죠 java는 + 가 되니까요
 //%f의 경우엔 (float, double)등을 처리합니다
 //printf("numberArray[%d] = %d₩n", i, numberArray[i])
 //위 케이스에서 첫 번째 %d와 두 번째 %d가 보입니다.
 //첫 번째는 ', i' 가 대응해서 i 값이 %d를 대체하게 됩니다.
```

```
        //두 번째는 ', numberArray[i]' 가 대응해서
        //numberArray[i]에 해당하는 배열값이 %d를 대체합니다.
        System.out.printf("numberArray[%d] = %d₩n",
                                    i, numberArray[i]);
        System.out.println("numberArray["+i+"]      =      "      +
                                    numberArray[i]);
}

    //아래와 같이 새로운 메모리를 할당해서 전달하는 것을 막고 있습니다.
    //조금 풀어보자면 객체를 상수화 하느냐
    //객체 내부의 값을 상수화 하느냐의 관점으로 봐야합니다.
    //현재 관점은 객체를 상수화하였기 때문에 다른 객체 대입이 막히는 모습입니다.
//numberArray = new int[8];

System.out.println();

for (final int num: numberArray) {
        System.out.println("numberArray elem: " + num);
```

```
        }
    }
}
```

10. 클래스 기본

```
package lectureClass;


//Led 클래스는 불이 켜졌다 혹은 꺼졌다만 관리하면 됩니다.
//클래스의 형태는 현재 아래와 같습니다.
/*
    ------ Led 객체 ------
    |        isTurnOn       |
    ---------------------
    |       Led 생성자       |   <<<- new를 하면 이와 같은 형태가 만들어지는 것입니다.
    |       getTurnOn       |
    |       setTurnOn       |
    ---------------------
 */
class Led {
```

```
private Boolean isTurnOn;


        //생성자는 class의 이름과 같습니다.
        //그리고 아래와 같이 리턴 타입이 없습니다.
        //만드는 규칙이라면 아래와 같습니다.
    //1. public을 적고 클래스 이름을 적은 이후 소괄호()
        를 작성후 중괄호{}를 작성합니다.
    //2. 만약에 외부에서 값을 입력 받을 것이라면 소괄호에
     입력받을 형태를 작성합니다.
   //3. 실제 클래스가 new를 통해 객체화 될 때 구동시키고
     싶은 작업을 중괄호 내부에 배치합니다.
   //즉 현재는 new를 통해 만들어질 때
   //기본값으로 전구를 꺼놓은 상태로 시작하게 됩니다.
public Led() {
    this.isTurnOn = false;
}


      //클래스 내부에 기능을 수행하는 집합들을 매서드라고
      부릅니다.
      //매서드를 작성하는 방법
      //1. public을 작성하고 소괄호()를 작성하고 중괄호를 {}
```

작성합니다.

//2. 리턴 타입을 public 옆에 작성합니다.

//3. 매서드의 이름을 그 옆에 작성해줍니다.

// 이름은 역시나 이 작업이 무엇을 하는지 명시적으로 알려줄 수 있는 형태가 좋습니다.

//4. 중괄호 내에서는 실제 매서드 이름에 해당하는 작업을 진행하면 됩니다.

// 이 때 단순히 작업만 하고 정보 반환이 없다면 리턴 타입은 void 입니다.

// 참/거짓을 반환한다면 Boolean(boolean) 입니다.

// 숫자등등이라면 Long(long)이나 Integer(int)입니다.

// 무엇을 반환(리턴) 하느냐에 따라 적절한 형태를 적어주면 됩니다.

// * 도대체 리턴 된다는 것은 무엇을 의미 하는가 ?

// 3 -> [ ] -> 9 (리턴 타입 int)

// 버튼 누름 -> [ ] -> true (리턴 타입 boolean)

// 1 -> [ ] -> "예금" (리턴 타입 String)

// 회원 정보 -> [ ] (리턴 타입 void) == 정확히는 리턴하지 않음을 의미함

// [ ] -> 20 (리턴 타입 int)

```
public Boolean getTurnOn() {
    return isTurnOn;
}

  //사실은 지가 지를 킬 수 없기 때문에 다른 객체의 도움을 받아야함
   //사실 수십년간의 SW 전문가들이 이러한 개념들을 정리하였고
   //그 개념의 일환으로 탄생하게 된 개념이 Domain Service 개념입니다.
  //Domain Service를 만들어서 얻는 이점은
  //비즈니스 관점을 좀 더 명확하게 만들어준다는 이점이 있습니다.
  //실제로 전구를 키고 끄는 작업을 가지고 비즈니스를 할만한 것들이 많지는 않지만
  //그래도 Domain Service를 나눠 본다면 아래와 같은 것들이 존재할 것입니다.
  //키기, 끄기, 깜빡이기

   // -> 조금 쉽게 접근해 보자면 Domain Service는
   // 객체 스스로가 직접 하기에는 표현이 애매해지는 작업
```

```
    //들을, 모두 Domain Service로 재배치하게 됩니다.
    //이를 통해 가독성과 유지보수성의 향상을 가져올 수 있습니다.

    //제어하는 Controller에 RequestForm 객체를 전달
    //RequestForm 객체는 Domain Service에서 처리하기 적합한 형태인 Request로 변환
    //그리고 Request를 보고 적절한 Entity를 추출하게 되는데
    //이런 관점으로 접근하면 setter를 완벽하게 제거할 수 있습니다.
    //즉 필요하면 final 객체에 final 변수들을 설정해서 전달한다는 의미입니다.

     // * 입력 관점에서 살펴봅시다!
     //          3 -> [       ] -> 9        (입력 타입 int)
     // 버튼 누름 -> [       ] -> true     (입력 타입 Button class)
     //          1 -> [       ] -> "예금" (입력 타입 int)
     //               [       ] -> 20       (입력 타입 void)
     //    참/거짓 -> [       ]             (입력 타입 boolean)
public void setTurnOn(Boolean turnOn) {
    isTurnOn = turnOn;
```

```
    }
}

public class LectureClassTest {
    public static void main(String[] args) {
        // OOP (Object Oriented Programming)
        // Domain Driven Development (DDD)

        //잘 만든 OOP란 무엇인가 ?
        //OOP란 모든 정보를 객체화하여 레고처럼
        //필요하면 조립하여 관리하자라는 뜻을 가지고 있습니다.

        //하지만 모든 정보를 하나의 클래스에서 관리하게 되는 경우
        //객체가 비대해지면서 해당 객체가 어떤 목적을 가지고 있었는지 목적성을 잃게 됩니다.
        //Domain이라는 관점은 이렇게 클래스가 어떤 주제에 집중을 하고 있는지를 본다고 생각하면 됩니다.
        //즉 내가 집중하는 주제가 무엇인가를 알 수 있도록 예쁘게 잘 표현해주는 것을 OOP라 봐도 무방합니다.
```

```
        // 전구(LED)를 키는 상황을 생각해봅시다.
        final Led led = new Led();


         System.out.println("현재 전구 상태: " + (led.getTurnOn() ?
                                        "켜짐" : "꺼짐"));
        led.setTurnOn(true);
        System.out.println("현재 전구 상태: " + (led.getTurnOn() ?
                                          "켜짐" : "꺼짐"));
    }
}
```