<문제은행 [1] _ 클래스화>

2. 정수형 변수 1개, float 변수 2개, double 변수 3개에 아무 값이나 할당하고 출력합니다.
3. 정수형 변수 2개에 랜덤값을 할당하고 출력합니다.
4. float 변수 2개에 랜덤값을 할당하고 출력합니다.
5. double 변수 2개에 랜덤값을 할당하고 출력합니다.

[MyCustomDataTypeTest(main) - MyCustomDataType]

- MyCustomDataTypeTest(main)

1. MyCustomDataType 클래스의 생성자 호출해 mcdt, mcdt2, mcdt3 객체 생성
new 클래스명( )은 객체화하는 작업, 이를 수행할 때 생성자가 등장
생성자는 메서드처럼 입력을 전달받을 수 있다.

```
MyCustomDataType mcdt = new MyCustomDataType( intNum: 1,  floatNum: 2,  doubleNum: 3);
MyCustomDataType mcdt2 = new MyCustomDataType( intNum: 1,  floatNum: 2f,  doubleNum: 3.0);
MyCustomDataType mcdt3 = new MyCustomDataType();
```

```
나는 (int, int, int) 생성자!
나는 (int, float, double) 생성자!
나는 기본 생성자!
```

2. allocRandom 메소드
MyCustomDataType 클래스의 allocRandom 메소드를 호출해
정수형 변수, float 변수, double 변수에 난수를 할당한다.

```
mcdt.allocRandom();
System.out.println(mcdt);

MyCustomDataType mcdt4 = new MyCustomDataType( intNum: 2, MyCustomDataType.INT_PROC);
mcdt4.allocIntRandom();
System.out.println(mcdt4);

MyCustomDataType mcdt5 = new MyCustomDataType( intNum: 2, MyCustomDataType.FLOAT_PROC);
mcdt5.allocFloatRandom();
System.out.println(mcdt5);

MyCustomDataType mcdt6 = new MyCustomDataType( intNum: 2, MyCustomDataType.DOUBLE_PROC);
mcdt6.allocDoubleRandom();
System.out.println(mcdt6);
```

- MyCustomDataType 클래스

1. 변수 선언

```
public class MyCustomDataType {
    int[] intArr;
    float[] floatArr;
    double[] doubleArr;

    int intRange;
    float floatRange;

    final int INT_MAX = 50;
    final int INT_MIN = 30;

    final float FLOAT_MAX = 4.75f;
    final float FLOAT_MIN = -4.75f;
    final int BIAS = 100;

    static final int INT_PROC = 1;
    static final int FLOAT_PROC = 2;
    static final int DOUBLE_PROC = 3;

    int intNum, floatNum, doubleNum;
```

각각의 자료형 난수를 저장할 3가지 배열 intArr, floatArr, doubleArr
정수형 범위를 저장할 intRange, 실수형 범위를 저장할 floatRange
정수형 난수의 최솟값과 최댓값을 의미하는 INT_MAX, INT_MIN
실수형 난수의 최솟값과 최댓값을 의미하는 FLOAT_MAX, FLOAT_MIN
실수형 난수 계산에 사용할 변수 BIAS
어떤 자료형의 난수를 발생시킬지 결정하는 변수 3가지 PROC
난수를 몇 개 발생시킬지 결정하는 변수 intNum

2-1. 객체 mcdt

- 생성자 만드는 방법
: public을 적는다.
 클래스 이름과 동일한 이름을 작성한다.
 메소드 작성하듯이 입력으로 사용할 정보들을 입력하도록 한다.
 중괄호 내부에 이 메서드(생성자)가 구동할 작업을 작성한다.
 + 생성자는 리턴 타입이 없다!

main에서 전달받은 intNum, floatNum, doubleNum에 따른 배열의 크기를 지정하고 각 자료형의 값들을 저장할 배열 3개를 생성한다.

setRange 메소드를 호출한다.

```
public MyCustomDataType (int intNum, int floatNum, int doubleNum) {
    System.out.println("나는 (int, int, int) 생성자!");

    intArr = new int[intNum];
    floatArr = new float[floatNum];
    doubleArr = new double[doubleNum];

    setRange();
}
```

- setRange : 난수값의 범위를 지정하는 메소드
int형 범위를 지정하는 setIntRange 메소드와
실수형 범위를 지정하는 setRealRange 메소드를 호출한다.

```
public void setRange () {
    setIntRange();
    setRealRange();
}
```

- setIntRange
intRange 값의 범위를 지정한다.

```
public void setIntRange () { intRange = INT_MAX - INT_MIN + 1; }
```

- setRealRange

실수형 값의 범위를 지정한다.

Math.random의 기본 범위는 0.0 ~ 1.0 미만이므로 .

-> Math.random() * floatRange을 했을 때 0.0~ 950.xxx까지의 난수가 발생되도록 951을 범위로 지정해야한다.

475 - (-475) = 950 + 1 = 951 //BIAS를 곱함

+ 난수 할당 시 BIAS(100)으로 다시 나누면 원하는 범위의 값을 얻어낼 수 있다.

-> 0.0 ~ 9.5xxx, 이 범위에 실수형 최솟값 FLOAT_MIN을 더해주면 -4.75 ~ 4.75xxx 사이의 난수를 얻을 수 있다.

```
public void setRealRange () {
    // 0.0 ~ 1.0 미만 - Math.random()
    // 4.75 + 4.75 -> 9.5
    // 0.0 ~ 9.5 미만 + 0.1
    // 0.1 ~ 9.6 미만 (9.5xx)
    // 그래서 최종적으로 ===> (-475 ~ 475) / 100
    // floatRange = 951
    floatRange = FLOAT_MAX * BIAS - FLOAT_MIN * BIAS + 1;
}
```

2-2. mcdt.allocRandom

각 자료형에 해당하는 난수를 발생시키는 메소드 3가지 호출

```
public void allocRandom () {
    allocIntRandom();
    allocFloatRandom();
    allocDoubleRandom();
}
```

- allocIntRandom

intArr 배열의 크기만큼 반복해서 난수를 발생시키고 인덱스에 순차적으로 저장

```
public void allocIntRandom () {
    // length는 실제 배열의 길이를 구해올 수 있음
    for (int i = 0; i < intArr.length; i++) {
        intArr[i] = (int) (Math.random() * intRange + INT_MIN);
    }
}
```

- allocFloatRandom

floatArr의 크기만큼 반복해서 난수를 발생시키고 인덱스에 순차적으로 저장

```
public void allocFloatRandom () {
    for (int i = 0; i < floatArr.length; i++) {
        // (0.0 ~ 950) + (-4.75f)
        //float tmp = ((int) (Math.random() * floatRange)) / BIAS + FLOAT_MIN;

        float tmp = (int) (Math.random() * floatRange);
        //System.out.printf("tmp = %f\n", tmp);

        tmp /= BIAS;
        //System.out.printf("tmp = %f\n", tmp);

        floatArr[i] = tmp + FLOAT_MIN;
    }
}
```

- allocDoubleRandom

doubleArr의 크기만큼 반복해서 난수를 발생시키고 인덱스에 순차적으로 저장

```
public void allocDoubleRandom () {
    for (int i = 0; i < doubleArr.length; i++) {
        double tmp = (int) (Math.random() * floatRange);
        tmp /= BIAS;

        doubleArr[i] = tmp + FLOAT_MIN;
    }
}
```

-출력 결과

```
나는 (int, int, int) 생성자!
나는 (int, float, double) 생성자!
나는 기본 생성자!
MyCustomDataType{intArr=[31], floatArr=[3.9499998, -2.33], doubleArr=[-2.59, 0.04000000000000036, -2.15]}
```

3. 특정 자료형의 난수만 발생시키는 객체 mcdt4, mcdt5, mcdt6

- main

```
MyCustomDataType mcdt4 = new MyCustomDataType( intNum: 2, MyCustomDataType.INT_PROC);
mcdt4.allocIntRandom();
System.out.println(mcdt4);

MyCustomDataType mcdt5 = new MyCustomDataType( intNum: 2, MyCustomDataType.FLOAT_PROC);
mcdt5.allocFloatRandom();
System.out.println(mcdt5);

MyCustomDataType mcdt6 = new MyCustomDataType( intNum: 2, MyCustomDataType.DOUBLE_PROC);
mcdt6.allocDoubleRandom();
System.out.println(mcdt6);
```

1) 생성자가 호출되었을 때 intNum의 값과 DECISION 값을 넘겨주며 객체를 생성한다.
  +) MyCustomDataType.DOUBLE_PROC
    = MyCustomDataType 클래스의 DOUBLE_PROC 변수
2) 지정된 자료형의 난수를 발생시킨다.
3) toString으로 맵핑된 객체의 정보를 출력한다.

- MyCustomDataType 클래스

```
static final int INT_PROC = 1;
static final int FLOAT_PROC = 2;
static final int DOUBLE_PROC = 3;
```

static으로 변수를 선언해 main에서도 이 값들을 사용할 수 있다.

1) 생성자

```
public MyCustomDataType (int intNum, final int DECISION) {
    System.out.println("나는 (int, int) 생성자!");

    System.out.println("DECISION: " + DECISION);
    decisionAlloc(intNum, DECISION);
}
```

생성자가 호출되었을 때 받아온 값들은 intNum과 DECISON이 되고,
decisionAlloc 메소드를 호출하면서 이 값들을 넘겨준다.

2) decisionAlloc

```java
public void decisionAlloc(int arrNum, final int DECISION) {
    switch (DECISION) {
        case INT_PROC:
            intArr = new int[arrNum];
            setIntRange();
            break;

        case FLOAT_PROC:
            floatArr = new float[arrNum];
            setRealRange();
            break;

        case DOUBLE_PROC:
            doubleArr = new double[arrNum];
            setRealRange();
            break;

        default:
            System.out.println("올바른 값을 입력하세요!");
            break;
    }
```

switch문을 이용해 받아온 DECISON 값에 해당하는 자료형의 범위를 설정한다.

3) alloc(int / float / double) Random

4) toString : 클래스의 정보를 맵핑시킨다.

```java
@Override
public String toString() {
    return "MyCustomDataType{" +
            "intArr=" + Arrays.toString(intArr) +
            ", floatArr=" + Arrays.toString(floatArr) +
            ", doubleArr=" + Arrays.toString(doubleArr) +
            '}';
}
```

- 출력 결과

```
나는 (int, int) 생성자!
DECISION: 1
MyCustomDataType{intArr=[31, 35], floatArr=null, doubleArr=null}
나는 (int, int) 생성자!
DECISION: 2
MyCustomDataType{intArr=null, floatArr=[-1.24, -0.7399998], doubleArr=null}
나는 (int, int) 생성자!
DECISION: 3
MyCustomDataType{intArr=null, floatArr=null, doubleArr=[2.9400000000000004, -3.9]}
```

[RandomGeneratorTest (main) - RandomGenerator]

9. 4 ~ 97까지의 랜덤 숫자를 생성해보세요.

- RandomGeneratorTest (main)

1. RandomGenerator 생성자 호출하며 객체 rg 생성
2. rg.confirmRandom() 메소드의 값이 false가 되었을 때 난수 생성 후 출력

```
RandomGenerator rg = new RandomGenerator( intMin: 4,  intMax: 97);

if (!rg.confirmRandom()) {
    System.out.println("난수 생성에 문제가 있음");
} else {
    System.out.println("난수 생성: " + rg.intGenerate());
}
```

- RandomGenerator

1. 변수 선언

```
int intMax, intMin, intRange;
final int CHECK_NUM = 100000;
```

최댓값, 최솟값, 난수의 범위, 난수발생 이상 여부를 확인하는 횟수

2. 생성자

```
public RandomGenerator (final int intMin, final int intMax) {
    this.intMin = intMin;
    this.intMax = intMax;

    intRange = intMax - intMin + 1;
}
```

4와 97을 받아와 이 객체의 intMin과 intMax에 저장한다.
난수 발생 범위를 계산한다.

3. confirmRandom

```java
public boolean confirmRandom () {
    int tmp;

    for (int i = 0; i < CHECK_NUM; i++) {
        tmp = intGenerate();

        if (tmp < intMin || tmp > intMax) {
            return false;
        }
    }

    return true;
}
```

CHECK_NUM 횟수만큼 난수를 발생시켜 임시값인 tmp에 저장한 뒤
지정한 범위를 벗어나면 false를 반환하고, 이상이 없다면 true를 반환한다.
난수를 발생시키는 구문은 intGenerate 메소드

4. intGenerate

```java
public int intGenerate () {
    return (int) (Math.random() * intRange + intMin);
}
```

지정한 범위의 난수값을 리턴한다.

- 출력 결과

```
난수 생성: 40
```

10. 65 ~ 90까지의 난수와 97 ~ 122까지의 난수를 무작위로 생성해봅시다.

- RandomGeneratorTest (main)

1. RandomGenerator 생성자 호출하며 객체 rg2 생성
2. rg2.confirmComplicatedRandom() 메소드의 값이 false가 되었을 때
   rg2.ComplicatedRandom() 메소드로 난수 생성 후 출력

```
RandomGenerator rg2 = new RandomGenerator(
        intMin: 65,  intMax: 90,  intMin2: 97,  intMax2: 122
);

if (!rg2.confirmComplicatedRandom()) {
    System.out.println("난수 생성에 문제가 있음");
} else {
    System.out.println("복합 난수 생성: " + rg2.complicatedRandom());
}
```

- RandomGenerator

1. 변수 선언

```
int intMax, intMin, intRange;
int intMax2, intMin2, intRange2;

final int TWO = 2;
final int CHECK_NUM = 100000;
```

첫 번째 범위에 대한 변수 : intMax, intMin, intRange
두 번째 범위에 대한 변수 : intMax2, intMin2, intRange2
percent50 메소드에 사용할 변수 TWO
난수발생 이상 여부를 확인하는 횟수 CHECK_NUM

2. 생성자

```
public RandomGenerator (
        final int intMin, final int intMax,
        final int intMin2, final int intMax2) {

    this.intMin = intMin;
    this.intMax = intMax;

    this.intMin2 = intMin2;
    this.intMax2 = intMax2;

    intRange = intMax - intMin + 1;
    intRange2 = intMax2 - intMin2 + 1;
}
```

main에서 값 4개를 받아와 두 개의 범위를 지정한다.

3. confirmcomplicatedRandom

```
public boolean confirmComplicatedRandom () {
    int tmp, tmp2;

    for (int i = 0; i < CHECK_NUM; i++) {
        tmp = intGenerate();

        if (tmp < intMin || tmp > intMax) {
            return false;
        }

        tmp2 = intGenerate2();

        if (tmp2 < intMin2 || tmp2 > intMax2) {
            return false;
        }
    }

    return true;
}
```

발생된 난수가 지정된 범위를 벗어나는지 확인하는 메소드로
tmp와 tmp2라는 임시변수에 발생된 난수를 저장해 이를 각각의 범위와 비교한다.
범위를 벗어났을 때만 false를 리턴해 난수발생에 문제가 있음을 알리고,
범위 안에 제대로 들어가면 true를 리턴해 난수를 출력시킨다.

3. complicatedRandom

```
public int complicatedRandom () {
    if (percent50() == 0) {
        return intGenerate();
    } else {
        return intGenerate2();
    }
}
```

if문의 조건을 percent50 메소드로 지정해 두 개의 범위 중 어떤 범위 안의 난수를 발생시킬지 결정한다.
0을 받으면 65 ~ 90까지의 난수를 발생시키는 intGenerate 메소드를 호출해 반환값을 리턴하고, 1을 받으면 97 ~ 122까지의 난수를 발생시키는 intGenerate2 메소드를 호출해 반환값을 리턴한다.

- percent50()

```
public int percent50 () {
    return (int) (Math.random() * TWO);
}
```

percent50 메소드에서 리턴하는 값은 0 or 1

- intGenerate, intGenerate2

```
public int intGenerate () {
    return (int) (Math.random() * intRange + intMin);
}
public int intGenerate2 () {
    return (int) (Math.random() * intRange2 + intMin2);
}
```

65 ~ 90까지의 난수를 발생시키는 intGenerate 메소드
97 ~ 122까지의 난수를 발생시키는 intGenerate2 메소드

- 출력 결과

```
복합 난수 생성: 80
```

11. 65 ~ 122까지의 난수를 무작위로 생성하고
65 ~ 90 혹은 97 ~ 122에 해당하는 숫자만 출력해봅시다.

- RandomGeneratorTest (main)

1. RandomGenerator 생성자 호출하며 객체 rg3 생성
2. rg3.conditionRandom() 메소드를 호출해 난수를 생성한 뒤 출력

```
RandomGenerator rg3 = new RandomGenerator(
        intMin: 65,  intMax: 122,
        condMin: 65,  condMax: 90,  condMin2: 97,  condMax2: 122
);

System.out.println("조건부 난수 생성: " + rg3.conditionRandom());
```

- RandomGenerator

1. 변수 선언

```
int intMax, intMin, intRange;

int condMin, condMax, condRange;
int condMin2, condMax2, condRange2;
```

- 65~ 122까지의 범위에 대한 변수
: intMax, intMin, intRange

- 65 ~ 90까지의 범위에 대한 변수
: condMin, condMax, condRange

- 97 ~ 122까지의 범위에 대한 변수
: condMin2, condMax2, condRange2

2. 생성자

```
public RandomGenerator (
        final int intMin, final int intMax,
        final int condMin, final int condMax,
        final int condMin2, final int condMax2) {

    this.intMin = intMin;
    this.intMax = intMax;

    intRange = intMax - intMin + 1;

    this.condMin = condMin;
    this.condMax = condMax;

    condRange = condMax - condMin + 1;

    this.condMin2 = condMin2;
    this.condMax2 = condMax2;

    condRange2 = condMax - condMin + 1;
}
```

main으로부터 값 6개를 받아와 전체범위, 부분범위1, 부분범위2를 설정한다.

3. conditionRandom()

```
public int conditionRandom () {
    int rand, cnt = 0;

    do {
        System.out.printf("%d 번째\n", ++cnt);
        rand = intGenerate();
    } while (isRandomNotOk(rand));

    return rand;
}
```

발생된 난수값을 담을 rand 변수와 발생된 난수 이상여부 확인횟수에 대한 변수인
cnt 변수를 선언하고 0으로 초기화한다.
do_while문을 통해 발생된 난수가 정확한 범위 내에 들어갈 때까지 반복해서 검사한다.
do에서 발생된 난수는 rand에 저장되고, while의 조건문에 위치한 isRandomNotOk
메소드에 넘어간다. isRandomNotOk 메소드에서 난수값에 대한 검사를 수행하며
true가 반환되면 난수발생, 검사를 다시 수행하며 false가 반환되면 반복이 종료되며
rand값을 리턴한다.

-isRandomNotOk

```
public boolean isRandomNotOk (int rand) {
    if ((rand >= condMin && rand <= condMax) ||
            (rand >= condMin2 && rand <= condMax2)) {
        return false;
    }

    return true;
}
```

발생된 난수 rand가 지정된 범위에 해당하는지 확인하는 메소드

- intGenerate

```
public int intGenerate () {
    return (int) (Math.random() * intRange + intMin);
}
```

65 ~ 122까지의 난수를 발생시키는 메소드

- 출력 결과

```
1 번째
조건부 난수 생성: 102
```

6. 주사위 2개를 굴려서 눈금의 합을 출력해봅시다.
7. 주사위를 굴려서 짝수인 경우 당첨입니다! 출력
8. 주사위를 굴려서 홀수가 나오면 손모가지를 내놔라 출력

[DiceGameTest(main) - DiceManager - Dice]

- DiceGameTest(main)

1. DiceManger 형식의 객체인 dm을 생성하고 diceNum값을 넘겨준다.
2. dm객체의 playDiceGame 메소드를 호출한다.
3. toString으로 맵핑된 dm의 정보를 출력한다.

```java
public class DiceGameTest {
    public static void main(String[] args) {
        // 주사위 2개를 굴려서 눈금의 합을 출력해봅시다.
        // 주사위의 개수 자체가 확장이 가능함
        DiceManager dm = new DiceManager( diceNum: 2);

        dm.playDiceGame();
        System.out.println(dm);
    }
}
```

- DiceManager

1. 변수 선언

```java
int diceNum;

Dice[] diceArr;
int sum;
```

주사위의 개수를 의미하는 diceNum
Dice 객체(각 주사위)를 저장할 배열 diceArr
주사위의 합을 저장할 sum

2. 생성자

```
public DiceManager (int diceNum) {
    sum = 0;
    this.diceNum = diceNum;

    diceArr = new Dice[diceNum];
}
```

main으로부터 diceNum값을 받아와 DiceManager클래스의 diceNum값에 저장한다.
Dice형식의 배열 diceArr의 크기를 diceNum으로 지정한 뒤 배열을 생성한다.

3. playDiceGame

```
public void playDiceGame () {
    int tmp;

    for (int i = 0; i < diceNum; i++) {
        // 주사위 객체 생성
        diceArr[i] = new Dice();
        // 주사위를 굴려야함
        // 합산
        tmp = diceArr[i].rollDice();
        System.out.printf("tmp = %d\n", tmp);
        sum += tmp;
    }

    checkWin(sum);
}
```

주사위 값을 저장할 임시변수 tmp를 선언한다.
for문을 통해 diceNum만큼 주사위게임을 반복한다.
diceArr 배열에 Dice객체가 순차적으로 저장된다
-> 각각의 주사위 객체에 각 주사위가 뽑은 값이 저장되도록 한다.

```
/* 메모리 영역

-------------------------------------
| Dice 객체 1 주소 | Dice 객체 2 주소 |    diceArr
-------------------------------------
        [0]              [1]

   각각의 Dice 객체는 아래와 같음
    --------------    --------------
    | MAX        |    | MAX        |
    | MIN        |    | MIN        |
    | range      |    | range      |
    -------------|    -------------|
    | Dice()     |    | Dice()     |
    | rollDice() |    | rollDice() |
    --------------    --------------
```

diceNum의 수만큼 생성된 객체에서 순차적으로 Dice 클래스의
1~6까지의 주사위값을 받아오는 rollDice 메소드를 실행시켜
tmp에 저장한 뒤 sum에 tmp값을 더한다.
반복이 종료되면 checkWin 메소드를 실행시켜 넘겨준 sum값을 통해
게임의 결과를 확인한다.

4. toString

```
@Override
public String toString() {
    return "DiceManager{" +
            "diceArr=" + Arrays.toString(diceArr) +
            ", sum=" + sum +
            '}';
}
```

DiceManager의 정보와 Dice객체들이 할당된 diceArr의 정보를 맵핑시킨다.

```
DiceManager{diceArr=[Dice@7f560810, Dice@69d9c55], sum=6}
```

- Dice

1. 변수 선언

```
final int MAX = 6;
final int MIN = 1;

int range;

public Dice () {
```

주사위의 범위 계산에 필요한 변수들을 선언한다.

2. 생성자

```
public Dice () {
    System.out.println("나는 Dice 클래스의 기본 생성자!");
    range = MAX - MIN + 1;
}
```

기본생성자로 호출될 때 주사위값의 범위를 계산한다.

3. rollDice

```
public int rollDice () { return (int) (Math.random() * range + MIN); }
```

호출되면 지정된 범위(1~6)의 난수값을 발생시켜 리턴한다.

- 출력 결과

```
나는 Dice 클래스의 기본 생성자!
tmp = 4
나는 Dice 클래스의 기본 생성자!
tmp = 3
올땐 쉬웠지 갈땐 손모가지닷!
```

```
나는 Dice 클래스의 기본 생성자!
tmp = 2
나는 Dice 클래스의 기본 생성자!
tmp = 4
당첨입니다!!!
```