2022.1.21 복습(24일차) 자바 마지막날(개어려움)

어제 복습했던 것. 기억이 나니. 문제점 : 코드가 너무 난잡했다.

```
public class Bank6Prob1 {
    public static void main(String[] args) throws InterruptedException {

        ThreadRectangle.calcEachThreadTotal();

        ThreadRectangle[] rect = new ThreadRectangle[ThreadRectangle.THREAD_MAX];

        // 스레드 준비
        for (int i = 0; i < ThreadRectangle.THREAD_MAX; i++)
        {
            rect[i] = new ThreadRectangle();
        }

        // 스레드 구동
        for (int i = 0; i < ThreadRectangle.THREAD_MAX; i++)
        {
            rect[i].run();
            rect[i].join();
        }

        // test 구동 (컴퓨터 세계의 오차의 모순을 확인)
        //rect[0].run();
        System.out.printf("%d개의 스레드가 모든 작업을 완료하였습니다.", ThreadRectangle.THREAD_MAX);
```

```
        float finalResult = 0;

        for (int i = 0; i < ThreadRectangle.THREAD_MAX; i++) {
            finalResult += rect[i].getSum();
        }

        System.out.println("최종 결과는 ? " + finalResult);
    }
}
```

변한코드 잠깐보기 (위 : 변경 전 / 아래: 변경 후)

```
public class Bank6ProbIntegration {
    public static void main(String[] args) throws InterruptedException {

        ThreadManager tm1 = new ThreadManager( threadNum: 6,  parameter1: 2,  parameter2: 2, ThreadCalculation.SQUARE);
        System.out.println("계산된 값 = " + tm1.calcArea());

        ThreadManager tm2 = new ThreadManager( threadNum: 4,  parameter1: 0,  parameter2: 3, ThreadCalculation.QUADRATIC);
        System.out.println("계산된 값 = " + tm2.calcArea());
    }
}
```

간단해졌음. 확실히. 코드를 보러 가자

ThreadManager 라는 클래스에서 쓰레드 관리하는 식으로 클래스를 만들었다.

생성자 안에 숫자를 넣기로 한다.

```java
public class ThreadManager {
    private ThreadRectangle[] tRect;
    // BlaBla[] blabla;
    private ThreadQuadraticEquation[] tqe;

    private int threadNum;
    private int serviceCode;

    // 4, 2, 2, ThreadCalculation.SQUARE
    public ThreadManager (int threadNum, int parameter1,
                          int parameter2, int serviceCode) {

        this.threadNum = threadNum;
        this.serviceCode = serviceCode;

        checkService(threadNum);
        serviceAlloc(parameter1, parameter2);
    }

    public void checkService (int threadNum) {
        switch (serviceCode) {
            case ThreadCalculation.SQUARE:
                tRect = new ThreadRectangle[threadNum];
                break;

            case ThreadCalculation.QUADRATIC:
                tqe = new ThreadQuadraticEquation[threadNum];
                break;
        }
    }
```

각 클래스마다 객체배열을 만들어줬다. (사각형 / y =x^2의 클래스)

threadNum / servoiceCode 는 이 클래스에서 변수를 입력해줬음.

아마도 매니저에서 해결해야하는 숫자들이기 때문일것같음.

생성자에서 값 받아서 각 매소드에 값을 보냄.

checkService 매소드에threadNum입력함 (사실 안넣어도 됨)

→해당 메소드 안에 이미 값을 입력했기 때문에 문제없음

serviceAlloc매소드(파라미터1,2는 넣어야함)

아무튼 checkService를 생성 → 각 serviceCode를 체크하기

SQUARE → 각 배열 생성 → threadNum숫자만큼의 크기로 만들기

QUARDTIC → 각 배열 생성 → threadNum숫자만큼의 크기로 만들기

```java
public void serviceAlloc (int parameter1, int parameter2) {
    switch (serviceCode) {
        case ThreadCalculation.SQUARE:
            ThreadCommon.threadNum = threadNum;
            ThreadCommon.calcRealTotal(parameter1);
            ThreadCommon.threadCnt = 0;

            for (int i = 0; i < threadNum; i++) {
                tRect[i] = new ThreadRectangle(threadNum, parameter1, parameter2);
            }
            break;

        case ThreadCalculation.QUADRATIC:
            ThreadCommon.threadNum = threadNum;
            ThreadCommon.calcRealTotal( x: parameter2 - parameter1);
            ThreadCommon.threadCnt = 0;

            for (int i = 0; i < threadNum; i++) {
                tqe[i] = new ThreadQuadraticEquation(threadNum, parameter1, parameter2);
            }
            break;
    }
```

For문으로 각 배열에 값 입력해주기

생성자에서 호출했던 serviceAlloc매소드도 가보자. 여기서는

파라미터 두개를 값에 대입했다 (이거는해야함, 클래스 안에 초기화 안함)

저 빨간것들은 왜 저기에 있냐, 선생님 왈.

저것들은 굳이 for문으로 반복 안해도 되는 것들이다.

threadNum = 어차피 몇 개로 나눌지 메인에서 정하기 때문에 굳이 반복을 할 필요가 없다.

calcRealTotal → 어차피 총 total은 x*0.001//반복해도 값은 똑같기 때문에 굳이 반복을 할 필요가 없다.

threadCnt → 이건 0부터+1해야하는데, 여기서 0으로 초기화 안해주면, 다른 값에서 영향을 받을 수 있음 (실제로 값도 확인했었음)

SQUARE – 0,1,2,3으로 값이 나왔다면, QUA~도 0,1,2,3이 나와야하는데, QUA가 4,5,6,7로 나왔었음. →그래서 밖에서 초기화해주기

```java
public class ThreadCalculation {
    static final int SQUARE = 0;
    static final int QUADRATIC = 1;
}
```

.

(옆에 클래스는 참고용 / 각 sqare, quar~이 stataic변수로 어디서든 해당 값을 이용 할 수 있도록 순서 정함.)

(계속되는 쓰레드 매니져분석)

```
public void eachThreadStartWork () {
    switch (serviceCode) {
        case ThreadCalculation.SQUARE:
            for (int i = 0; i < threadNum; i++) {
                tRect[i].start();
            }
            break;

        case ThreadCalculation.QUADRATIC:
            for (int i = 0; i < threadNum; i++) {
                tqe[i].start();
            }
            break;
    }
}

public void eachThreadWaitFinish () throws InterruptedException {
    switch (serviceCode) {
        case ThreadCalculation.SQUARE:
            for (int i = 0; i < threadNum; i++) {
                tRect[i].join();
            }
            break;

        case ThreadCalculation.QUADRATIC:
            for (int i = 0; i < threadNum; i++) {
                tqe[i].join();
            }
            break;
    }
}
```

각각의 쓰레드 작업을 시작한다.

Start를 사용하여 값을 실행하게 한다.

(For문으로 각 객체배열의 값을 도출한다.)

각각의 쓰레드 작업을 홀드한다.

Join을 사용하여 main의 값이 끝나기 전까지 값을 없애지 않는다.

(for문으로 모든 배열의 값을 홀드 할 수 있도록 한다.)

```
public float sumEachThreadResult () {
    float sum = 0;

    switch (serviceCode) {
        case ThreadCalculation.SQUARE:
            for (int i = 0; i < threadNum; i++) {
                sum += tRect[i].getSum();
            }
            break;

        case ThreadCalculation.QUADRATIC:
            for (int i = 0; i < threadNum; i++) {
                sum += tqe[i].getSum();
            }
            break;
    }

    return sum;
}

public float calcArea () throws InterruptedException {
    eachThreadStartWork();
    eachThreadWaitFinish();
    return sumEachThreadResult();
}
```

결과를 sum한다.

For문 통해 각 객체배열의 값을 순차적으로 구한다.

<- 결과 값 출력해주는 매소드

매소드 명을 보면 각각 어떤 일을 하는지 명확하게 알 수 있다.

내가 코드 짤때도 이런식으로 만들어야한다!!

매니저 클래스 끝, 이제 세부 클래스 보러가자.

1, ThreadRectangle 클래스

```java
public class ThreadRectangle extends ThreadCommon {
    // 1. main()쪽에서 ThreadRectangle tRect = new ThreadRectangle();
    //    tRect.calcArea(범위); 형식으로 처리하는 것을 원함
    // 2. ThreadRectangle 생성자가 구동될 때
    //    사용자가 요청한 스레드 개수만큼 스레드를 처리할 배열을 만들어야 한다.
    // 3. 이렇게 정리를 해놓고보니 가독성이 떨어지는 부분이 어딘가 ?

    // 이걸 몇 개의 Thread로 연산할거니 ?
    // 에 대한 답을 받아오도록 설계한다.
    public ThreadRectangle (int threadNum, int x, int y) { super(x, y); }

    @Override
    public void run() {
        for (int i = xStart[localThreadId]; i <= xEnd[localThreadId]; i++) {
            sum += dx * y;
            System.out.printf("Thread ID = %d, sum = %.12f\n", localThreadId, sum);
        }

        System.out.printf("sum = %.12f\n", sum);
    }
}
```

잘보면 여기서는 생성자로부터 super(부모클래스)에 값을 전달하고

Run을 통해 쓰레드 작성을 한 것 빼고는 다른 일을 하고 있지 않다.

모두 부모 클래스에서 작업 할 수 있도록 코드를 단순화 시켰다.

2, ThreadQuadraticEquation 클래스

```java
public class ThreadQuadraticEquation extends ThreadCommon {

    public ThreadQuadraticEquation(int threadNum, int x, int y) { super(x, y); }

    @Override
    public void run() {
        float curX = dx * xStart[localThreadId];

        for (int i = xStart[localThreadId]; i <= xEnd[localThreadId]; i++, curX += dx) {
            sum += dx * curX * curX;
            System.out.printf("Thread ID = %d, sum = %.12f\n", localThreadId, sum);
        }

        System.out.printf("sum = %.12f\n", sum);
    }
}
```

여기서도 생성자로부터 부모 클래스에 값을 전달하는 것과

Run을 통해 쓰레드 실행 말고는 다른 일을 하고 있지 않다.

부모 클래스에서 작업을 넘기고, 코드를 단순화 시켰다.

두개의 클래스에는 비슷한 느낌이지만,, 각각 구해야하는 값과 식이 조금씩 다르다.

1번은 사각형 구하는 2*2 // 2번은 y = x^2 그렇기 때문에 기본적인 사항을 제외하고는

계산식은 구분해서 run통해 값을 구할 수 있게 만들었다.

여기서 주의점.--> xStart를 배열로 만들어서 값을 [localThreadid]로 했는데 값이 잘 들어간것같다.

어떻게 잘 들어갔는지에 대해서는 값을 넘긴 부모 클래스로 가야할 것 같다.

(드디어 부모 클래스 입장)

사실 둘이 기초 값을 구하는데는 비슷한 점이 많다.

1. 0.001로 나눴을때의 x의 total값 구하기
2. total값을 thread사용을 위해 n개로 나눌지 구하기
3. ex)2000개/n:4 = 500기준// 첫번째 = 0~499 / 두번째 = 500~999 / 세번째 = 1000~1499 이런식으로 나눈 후 값을 배열에 넣어주기.

구하는 방식이 비슷하기 때문에 해당 값을 부모클래스로 만들어서 두 클래스가 해당 클래스를 상속하도록 한다.

```java
public class ThreadCommon extends Thread {

    final static float dx = 0.001f;
    static int threadCnt = 0;
    protected int localThreadId;

    protected int total;

    protected float sum;

    protected int x, y;

    protected static int realTotal;
    protected static int threadNum;
    protected static int totalRemain;
    // 현재 이녀석은 Critical Section이 되어버림
    // 해결책은 2가지
    // 1. Lock을 걸거나
    // 2. 배열로 만들어서 영향을 받지 않게 함 <<<--- 이게 효율적임 (스레드 개수만큼 할당해야함)
    // Critical Section은 여러 태스크(스레드, 프로세스)에 의해 공유되는 자원
    protected static int[] xStart;
    protected static int[] xEnd;

    // 가로 길이 2를 아주 작은값인 0.001로 나눈다.
    // 여기서 스레드를 4개 정도 사용한다면
    // 0 ~ 2를 4등분 하고 각각의 스레드에게 이것을 4등분해서 주면 된다.
```

자식클래스에서 변수 값을 생성한다. (은근 헷갈리고 중요)

Dx = 변동없는 x값(또는 작은 밑변)이기 때문에 final static

threadCnt = 매니저클래스에서 0초기화 해야해서 static

total, sum, x, y는 해당 클래스 내에서 사용하니 protected

realTotal, ThreadNum, totalRemain은 static인 이유

→calcRealTotal클래스가 매니저 클래서에 사용되서

(참고로 calcRealTotal클래스도 static으로 선언했다.)

xStart[], xEnd[] 또한 calcRealTotal에서 사용

+ 각 자식 클래스에서도 sum값 구하기 위해 사용하기 위해 static

```java
    // 1. 등분 했을 경우 스레드 각각이 돌려야 하는 전체 숫자를 먼저 파악
    // 2. 이것을 베이스로 시작과 끝 값을 정하면 됨
    public ThreadCommon(int x, int y) {
        this.x = x;
        this.y = y;

        // 1. realTotal값을 통해서 threadNum으로 나눴을때 몫이 얼마가 나오는지 판정한다.
        // 2. 나머지값을 판정한다.
        // 3. 각 threadId 들에게 나머지가 0이 될 때까지 1개씩 나눠준다.
        calcEachThreadTotal();

        localThreadId = threadCnt++;

        // xTotalEnd는 전역으로 현재 어디까지 갔는지 파악
        // xEnd는 실제 local(지역) 변수로 스레드마다 각기 다르게 가지게 해야함
        if (localThreadId == 0) {
            xStart[localThreadId] = 0;
            xEnd[localThreadId] = total - 1;
        } else {
            //          [0]     [1]     [2]     [3]     [4]     [5]
            // total    334     334     333     333     333     333
            // xStart     0     334     668    1001    1334    1667
            // xEnd     333     667    1000    1333    1666    1999
            xStart[localThreadId] = xEnd[localThreadId - 1] + 1;
            xEnd[localThreadId] = xEnd[localThreadId - 1] + total;
        }

        sum = 0;

        System.out.printf("xStart = %4d, xEnd = %4d, thread ID = %d\n",
                xStart[localThreadId], xEnd[localThreadId], localThreadId);
```

기본 생성자 생성 (자식 클래스에서 x,y값을 가지고 왔다.)

X,y 값을 해당 클래스 내에서 자유롭게 사용할 수 있게 this사용

(값을 구하기 위해 total값을 구하는 것이 먼저이기에 calcRealTotal 매소드를 먼저 사용하도록 한다.)

(realtotal은 아까 매니저에서 실행 완료해서 값을 알고있음)

localThreadId통해 +1을 해서 n번 나누는지 표현

if(Id=0이면→해주는 이유. 이거 안하고 해봤는데 안되더라.

안되는 이유는 xEnd[id-1]에서 -1이 안됨. 배열 0번째인데 -1하면 오류가 난다..! 그래서 0번째는 그냥 미리 값을 넣어줌.

Id가 1이상부터는 값을 제대로 표현할 수 있음.

저 값은 calcEachTreadTotal의 내용을 보면 더 잘 이해할 수 있음.

그리고 sum 초기화 (=0) 통해 클래스내에서 모두 사용 가능하도록 값을 초기화 해줌

(부모클래스의 total 값 구하는 매소드 구경하기)

```
public static void calcRealTotal (int x) {
    realTotal = (int) (Math.ceil(x / dx));
    System.out.println("realTotal = " + realTotal);

    totalRemain = realTotal % threadNum;
    xEnd = new int[threadNum];
    xStart = new int[threadNum];

    for (int i = 0; i < threadNum; i++) {
        xStart[i] = 0;
        xEnd[i] = 0;
    }
}

public void calcEachThreadTotal () {
    total = realTotal / threadNum;

    if (totalRemain-- > 0) {
        total++;
    }

    System.out.println("total = " + total);
}

public float getSum() { return sum; }
```

나는 솔직히 여기가 제일 어려웠음 (부모클래스 생성자도)

리얼토탈 클래스는 → 아까 얘기했듯 static으로 처리한다.

매니져클래스에서 해당 값을 사용하기 때문 (반복처리 안해도됨)

그리고 각 xStart와 xEnd값에 배열을 만든다.

(원래 안만들려고 하신것같은데, 아무래도 각 값을 **"순차적으로 중복없이"** 대입하고 넣고 하기엔 배열을 만드는게 더 효율적인였던 것 같음)

그리고 각 xS,xE 배열값이에 0으로 초기화해줌. 오? 이건 왜 하신거지?? 그냥 초기화 해주는 과정인건가?

해당 매소드는 각 토탈이 얼마인지 계산해준다.

중요한점 = 나눈 후의 나머지는?? -> 누락될 것임.

이것땜에 if문 사용

totalRemain이 나눈값의 나머지인데, 만약 나머지가 있다면 total에 값을 하나씩 더해주면서 나머지값을 하나씩 빼는 방법이다.(천재)

결론 : 위의 값을 구한 후,

해당 매소드의 TOTAL 값으로

옆에있는

xStart 와 xEnd 값을 구해야한다는 뜻

```
// 1. 등분 했을 경우 스레드 각각이 돌려야 하는 전체 숫자를 먼저 파악
// 2. 이것을 베이스로 시작과 끝 값을 정하면 됨
public ThreadCommon(int x, int y) {
    this.x = x;
    this.y = y;

    // 1. realTotal값을 통해서 threadNum으로 나눴을때 몫이 얼마가 나오는지 판정한다.
    // 2. 나머지값을 판정한다.
    // 3. 각 threadId 들에게 나머지가 0이 될 때까지 1개씩 나눠준다.
    calcEachThreadTotal();

    localThreadId = threadCnt++;

    // xTotalEnd는 전역으로 현재 어디까지 갔는지 파악
    // xEnd는 실제 local(지역) 변수로 스레드마다 각기 다르게 가지게 해야함
    if (localThreadId == 0) {
        xStart[localThreadId] = 0;
        xEnd[localThreadId] = total - 1;
    } else {
        //          [0]     [1]     [2]     [3]     [4]     [5]
        // total    334     334     333     333     333     333
        // xStart     0     334     668    1001    1334    1667
        // xEnd     333     667    1000    1333    1666    1999
        xStart[localThreadId] = xEnd[localThreadId - 1] + 1;
        xEnd[localThreadId] = xEnd[localThreadId - 1] + total;
    }

    sum = 0;

    System.out.printf("xStart = %4d, xEnd = %4d, thread ID = %d\n",
            xStart[localThreadId], xEnd[localThreadId], localThreadId);
```