디지털컨버전스 기반 UXUI Front 전문 개발자 양성과정

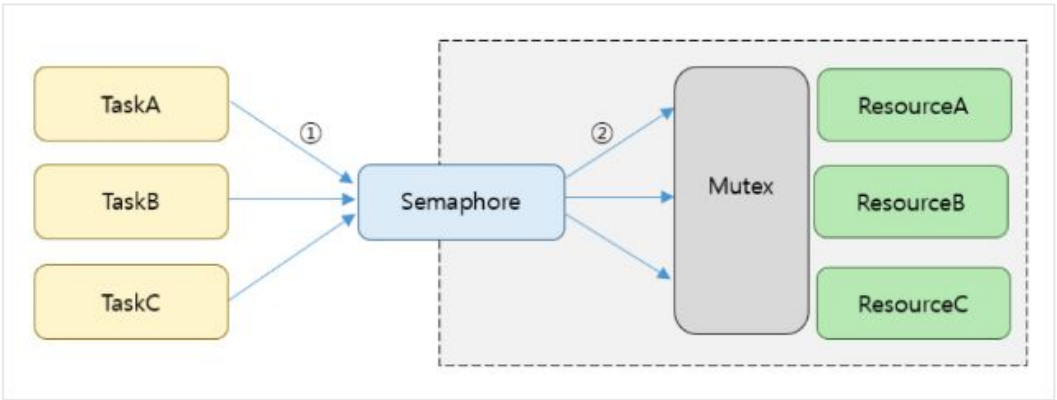
강사 - Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 - JungHyun LEE(이정현)
akdl911215@naver.com

뮤텍스란 무엇인가?
Mutex는 Mutual exclusion(상호배제)의 약자로, 임계 구역(Critical Section)문제를 해결할 수 있는 개발 도구 중 하나이다.

일종의 Locking 매커니즘이다. lock을 가지고 있을때만 공유 데이터에 접근이 가능하다. 화장실에 갈 때 키를 가진 사람만이 갈 수 있다. 일을 다 본후에는 키를 반납하고 그 다음 사람이 갈 수 있다. 이러한 메커니즘을 말한다. 유의할 점은 lock에 대한 소유권이 있다는 점. 열쇠를 획득한 사람만 반납 할 수 있다.

세마포어란 무엇인가?
세마포어는 동시에 리소스에 접근 할 수 있는 "허용 가능한 counter의 개수"를 말한다. 예를 들면, 병원에 있는 어느 한 병실에 5명까지 들어 갈 수 있다고 한다면, 5명까지 들어 갈 때 countiong을 하고 5명 이후에는 밖에서 기다려야 한다. 한 사람이 나오게 되면 다음 사람이 들어갈 수 있게 되는 것이다.

그리고 count수에 따라서 1개이면 binary semaphore, 여러 개이면 counting semaphore라고 한다. binary semaphore는



레이스 컨디션(race condition)이란 무엇인가?
경합조건이라고 하며, 유닉스 계열의 시스템에서 여러개의 프로세스가 동시에 실행될 때, 서로 CPU에 대한 우선권을 차지하기 위해 경쟁하게 되는데, 그것이 바로 레이스 컨디션이다. 이것은 규칙적인 것이 아니라 불규칙하게 일어난다. 이를 이용해 해킹에 사용되기도 한다.

동기화(Syschronezed)
둘 이상의 스레드가 공동의 자원(파일이나 메모리 블록)을 공유하는 경우, 순서를 잘 맞추어 다른 스레드가 자원을 사용하고 있는 동안 한 스레드가 절대 자원을 변경할 수 없도록 해야 한다. 한 스레드가 파일에서 레코드를 수정하는데, 다른 스레드가 동시에 같은 레코드를 수정하면 심각한 문제가 발생할 수 있다. 이런 상황을 처리할 수 있는 한 방법은 관련된 스레드에 대한 동기화를 이용하는 것이다.

동기화의 목적은 여러 개의 스레드가 하나의 자원에 접근하려 할때 주어진 순간에는 오직 하나의 스레드만이 접근 가능하도록 하는 것이다. 동기화를 이용해 스레드의 실행을 관리할 수 있는 방법은 두 가지가 있다.

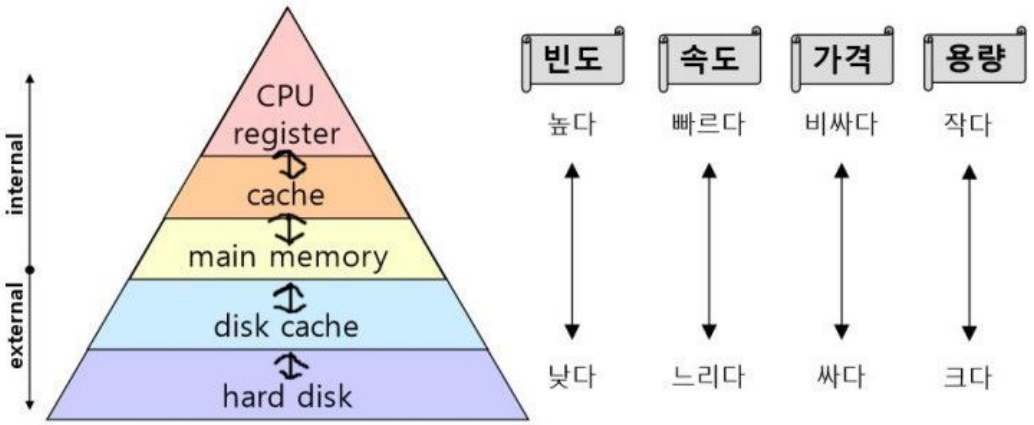
- 코드를 메소드 수준에서 관리 할 수 있다. - 동기화 메소드
- 코드를 블록 수준에서 관리 할 수 있다. - 동기화 블록

동기화 메소드와 동기화 블록은 모두 sychronized를 이용하여서 구현된다. 파워풀한 기능을 간단하게 사용할 수 있다는 장점이 있지만 남발하면 안됩니다. 성능상에 문제를 줄 수 있기 때문입니다.

크리티컬 섹션(Critical Section)
운영체제가 지원하는 동기화 방법의 하나로 크리티컬 섹션은 번역하자면 '임계 구역'혹은 '치명적 영역'이라고 할 수 있다. 치명적 영역은 보호되어야 하듯이, 이 역시 보호되어야 할 영역에 이른다. '공유 자원의 독점을 보장해주는 역할을 수행한다.

한마디로 요약하면, 한 순간에 하나의 스레드만 접근이 요구되는 공유 자원에 접근 하는 코드 영역을 의미한다. 이러한 문제가 생기는 이유는 각 스레드들은 각자의 Stack과 Register만 독립적으로 갖고 있고, 나머지 자원들은 '공유'하고 있기 때문이다

메모리 계층 구조(Memory hierarchy)
메모리 계층 구조란 메모리를 필요에 따라 여러가지 종류로 나누어 둠을 의미한다. 이때 필요한 대부분의 경우 CPU가 메모리에 더 빨리 접근하기 위함이다. 일반적으로 레지스터와 캐시는 CPU내부에 존재한다. 당연히 CPU는 아주 빠르게 접근할 수 있다. 메모리는 CPU 외부에 존재한다. 레지스터와 캐시보다 더 느리게 접근 할 수 밖에 없다. 하드 디스크는 CPU가 직접 접근할 방법 조차 없다. CPU가 하드 디스크에 접근하기 위해서는 하드 디스크의 데이터를 메모리로 이동시키고, 메모리에서 접근해야 한다. 아주 느린 접근 방법 불가능하다



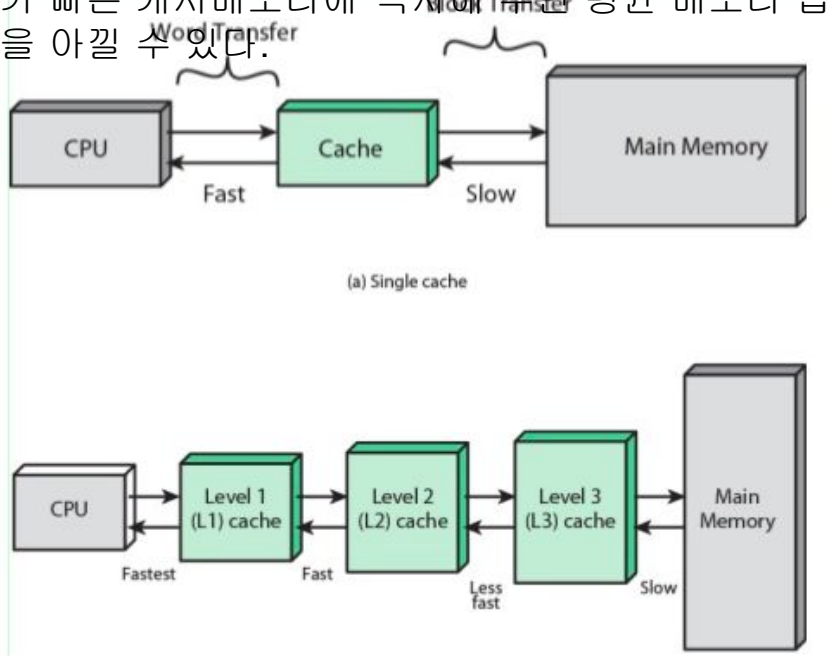
그렇다면 메모리 계층구조는 어째서 피라미드 형태로 말할까?
이는 컴퓨터 과학에서 증명된 법칙인데, 큰 메모리를 사용한다고 해도 그 안의 모든 데이터를 고르게 접근하지 않는다. 자주 쓰이는 데이터는 계속 자주 쓰이고, 자주 쓰이지 않는 데이터는 계속 자주 쓰이지 않기 때문이다. 쉽게말해서 멱법칙 인것이다.

이를 이용해서 운영체제나 CPU는 자동으로 자주 쓰이는 데이터, 또는 자주 쓰일 것 같은 데이터를 메모리에서 캐시로 읽어온다. 자주 쓰이는 데이터는 전체 데이터 양에 비해 작은 양이기 때문에, 캐시는 메모리보다 더 작아도 된다. 메모리와 하드 디스크의 관계도 마찬가지이다. 그리고 이러한 법칙을 토대로 경제성을 고려하여 만들어진것 일것이다

간단하게 메모리 구조의 상층으로 갈수록 비싸지기 때문이다. 비싼 하드웨어는 꼭 필요한 만큼만의 크기만 사용하고, 싼 하드웨어를 넉넉한 크기만큼 사용하기 때문에

레지스터 (Registers)
프로세서 레지스터 또는 단순히 레지스터는 컴퓨터의 프로세서 내에서 자료를 보관하는 아주 빠른 기억 장소이다. 일반적으로 현재 계산을 수행중인 값을 저장하는 데 사용된다.
레지스터는 용도에 따라 전용 레지스터와 범용 레지스터로 나뉜다. 컴퓨터에서 제일 빠른 메모리이며, CPU 계산 과정에서 작동한다.

캐시(Cache)
CPU 캐시는 메인 메모리에서 가장 자주 사용되는 위치의 데이터를 갖고 있는, 크기는 작지만 빠른 메모리이다. 대부분의 메모리 접근은 특정한 위치의 근방에서 자주 일어나는 경향이 있기 때문에, 데이터를 크기는 작지만 속도가 빠른 캐시메모리에 복사해 두면 평균 메모리 접근 시간을 아낄 수 있다.



프로세서가 메인 메모리를 읽거나 쓰고자 할 때는, 먼저 그 주소에 해당하는 데이터가 캐시에 존재하는지를 살핀다. 만약 그 주소의 데이터가 캐시에 있으면 데이터를 캐시에서 직접 읽고, 그렇지 않으면 메인 메모리에 직접 접근한다.

속도로는 레지스터 다음으로 빠른 메모리이며, 캐시 메모리도 속도와 용량에 따라 여러 단계로 나뉜다. L1 캐시, L2 캐시, L3 캐시 등으로 숫자가 작을수록 용량이 작고 빠르며, 숫자가 클수록 용량이 크고 느린 물건이다. CPU에 가까울수록 상위 레벨 캐시로 취급하며, 최하위 레벨 캐시의 경우 마지막 레벨을 의미하는 LLC(Last Level Cache)라고

캐시를 쉽게 비유하자면, 보통 도서관에서 공부를 할 때 우리는 필요한 책들을 쭉 뽑아서 책상위에 쌓아두고 공부를 시작합니다. 이렇게 하면 우리가 필요한 대부분의 정보들은 책상 위에서 빠르게 얻을 수 있고, 사실 도서관 전체에는 나와 관련이 없는 많은 자료들이 있는 것에 반해 책상위에 뽑아 놓은 책에는 내게 필요한 정보들이 존재하게 됩니다. 이렇게 필요한 책을 뽑아 우리가 필요할 만한 정보들을 주변에 배치해 놓는 것처럼 컴퓨터 에서도 우리가 자주 사용할 만한 정보들을 가까운 곳에 복사해 놓게 되고 이것을 우리는 캐시라고 합니다.

그럼 한정적인 공간인 캐시에 어떤식으로 선별이 될까? 지역성의 원칙(principle of locality)가 적용되기 때문이다. 금방 예를들었던 도서관에서 책을 뽑아오는 것과 같은 논리이며 지역성의 원칙에 따라 선별됩니다. 도서관에는 필요한 책들이 종류별로 분류되어 있고, 아마 내게 필요한 정보는 주변에 붙어 있을 확률이 높습니다. 내가 A라는 책을 가져오면서 혹시 몰라서 A 책 주변의 다양한 책들을 가져오는 것이 이러한 지역성의 원칙을 따른 캐싱이라고 볼 수 있습니다. 또는, 내가 예전에 한번 봤던 책들에 내가 필요한 정보가 담겨있을 확률도 높습니다. 위처럼 내가 필요한 정보 주변의 책에 필요한 정보가 있을 확률이 높은 것을 공간적 지역성이라고 하고, 또 예전에 내가 사용했던 책에 필요한 정보가 있을 확률이 높은 것을 시간적 지역성이라고 합니다. 이러한 두 가지 지역성을 원칙에 따라 우리는 책을 가져오고 컴퓨터에서는 캐싱을 통해 이러한 작업을 수행합니다.

캐시히트
CPU가 데이터를 요청할 경우 바로 메인 메모리에서 데이터를 가져오는 것이 아니라 일단 캐시에 데이터가 있는지 요청을 하게 되고 만약 있다면 캐시에서 바로 해당 데이터를 가져오는 것

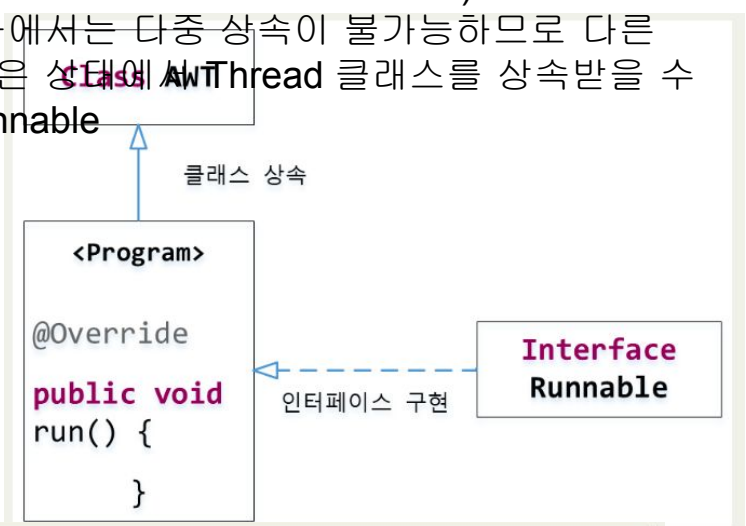
캐시미스
만약 캐시에 데이터가 없어서 다시 메인메모리에 데이터를 요청하게 되는 것

block(line)
캐시의 기본 저장 단위 ㄸ고는 메모리의 각 계층에서 데이터를 전송하는 기본 단위(multiple word) 현대 컴퓨터의 block size는 64bytes. 즉 블록 당 16개의 word를 담고 있음

hit : 접근한 데이터가 상위 계층에 있을때 (hit rate = hits/accesses)

miss : 접근한 데이터가 상위 계층에 없을 때 (miss rate = misses/accesses = 1 - hitrage) miss 발생 시 하위 계층에서 블록은 복사해옴

Runnable
-Runnal은 작업스레드가 실행할 수 있는 코드인 run() 메소드를 가지고 있는 객체라는 의미에서 able로 이름이 붙여짐.
-Runnable 인터페이스를 구현하는 클래스의 인스턴스 대상으로 Thread 클래스의 인스턴스를 생성하여 스레드를 생성합니다.
-이 방법은 상속할 클래스가 존재할 때 유용하게 사용된다.
(다른 클래스를 반드시 상속받아야 할 경우)
-정리하자면 자바에서는 다중 상속이 불가능하므로 다른 클래스를 상속받은 상태에서 Thread 클래스를 상속받을 수 없다. 그래서 Runnable 인터페이스를 이용하면 다른 클래스를 상속받더라도 스레드 구현이 가능하게 된다.



단계	설명
1	Runnable 인터페이스를 이용하여 스레드 클래스를 정의하고, run() 메서드 재정의
2	Runnable 인터페이스를 이용하여 정의된 스레드 클래스의 객체 생성
3	생성된 객체를 이용하여 start() 메서드 호출

Runnable은 인터페이스이기 때문에 구현 클래스를 만들어야 한다. Runnable에는 run() 추상메소드가 정의되어 있는데 구현 클래스에서 run()메소드를 재정의해서 작업 스레드가 실행할 코드를 작성하면 된다. Runnable은 작업 내용을 가지고 있는 객체이지 실제 스레드는 아닙니다. 그러므로 Runnable 구현 객체를 생성 한 후, 이를

FailedBabk

```java
public class FailedBank {
    // 10만을 할당
    private int money = 100000;

    public int getMoney() {
        return money;
    }

    public void setMoney(int money) {
        this.money = money;
    }

    // plus 값을 입력
    public void plusMoney(int plus) {
        // private int money = 100000; 을 m 에 할당
        int m = getMoney();

        try {
            // 슬립은 왜쓰는 건가요??????
            // millis를 위아래 두개를 10,10 / 100,10 일때는
            // 80, 50 일때와 값은 똑같고 0 , 0 으로 하면
            // 값이 플러스 388000, 마이너스 296000이 됩니다.
            // 아마 싱크로나이즈처럼 순서를 정해주기 위한 방식 같은데
            // 그렇다면 0,0일 경우에도 값이 이상해져야하는데 의문입니다
            Thread.sleep( millis: 80);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // plus 값을 입력
    public void minusMoney(int minus) {
        int m = getMoney();

        try {
            Thread.sleep( millis: 50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 할당 받은 m에다가 plub 값을 빼서
        // setMoney 로 리턴한다
        setMoney(m - minus);
    }
}
```

WhyThreadMutex (수업 21.01.26 nineteenth 내용)

```
// A 에다가 Thread를 상속
// Thread를 상속해줄 수 있는 이유는 A a = new A(); 또는
// a.start(); 을 선언했기때문이라고 생각이 됩니다.
class A extends Thread {
    public void run() {
        for(int i = 0; i < 100; i++) {
            WhyThreadMutex.fb.plusMoney(3000);
        }

        System.out.println("plusMoney(3000): " +
                WhyThreadMutex.fb.getMoney());
    }
}

class B extends Thread {
    public void run() {
        for(int i = 0; i < 100; i++) {
            WhyThreadMutex.fb.minusMoney(1000);
        }

        System.out.println("minusMoney(1000): " +
                WhyThreadMutex.fb.getMoney());
    }
}
```

```
public class WhyThreadMutex {
    // FailedBank 를 new 로 생성해서 가져온다.
    public static FailedBank fb = new FailedBank();

    public static void main(String[] args) {
        System.out.println("원금: " + fb.getMoney());

        A a = new A();
        B b = new B();

        a.start();
        b.start();

        // 사실상 이 문제를 해결하기 위해 도입해야 하는 것이
        // Lock Mechanism(Mutex, Semaphore, Spinlock) 이다.
        // 이들은 결론적으로 지금 거론한 화장실 사용중을 표기하는 것이다.
        // 자바는 스핀락 사용이 쉽지 않음
        // (일반적으로 스핀락 코드는 어셈블리어를 통해 구현됨)
    }
}
```

SynchronizedBankTest (수업 21.01.26 nineteenth 내용)

```
// 결과값을 보면 3000씩 계속올라가다가 어느순간에 1000씩
// 계속 줄어듭니다 그리고 올라갔다 내려갔다를 계속 반복합니다.
// 그럼 제가 혼자서 가정을 해봤을때 총 200만번(c와 d 반복)을 결과를
// 출력할때 일정 비율(갯수)마다 출력이 된다고 생각됩니다.
// 제 가정이 맞았을 경우 3가지가 궁금합니다.
// 1) 싱크로나이즈는 단순히 레이스컨디션을 일으키지 않게 도와주는
//    것이며 우선순위는 같을 경우에 랜덤으로 실행이 되는게 맞나요?
// 2) 일정 비율(갯수) 또한 랜덤인가요?
// 3) 싱크로나이즈는 우선순위가 같을때라도 무조건 먼저 실행하게
//    만드는 방법은 없나요?
class C extends Thread {
    public void run() {
        for(int i = 0; i < 1000000; i++) {
            SynchronizedBankTest.sb.plusMoney(3000);
        }

        //SynchronizedBankTest.sb.plusMoney(3000);

        System.out.println("plusMoney(3000): " +
                SynchronizedBankTest.sb.getMoney());
    }
}

class D extends Thread {
    public void run() {
        for(int i = 0; i < 1000000; i++) {
            SynchronizedBankTest.sb.minusMoney(1000);
        }

        //SynchronizedBankTest.sb.minusMoney(1000);

        System.out.println("minusMoney(1000): " +
                SynchronizedBankTest.sb.getMoney());
    }
}

public class SynchronizedBankTest {
    public static SynchronizedBank sb = new SynchronizedBank();

    // 예외는 InterruptedException 로 던져버린다.
    public static void main(String[] args) throws InterruptedException {
        System.out.println("원금: " + sb.getMoney());

        C c = new C();
        D d = new D();

        c.start();
        d.start();
    }
}
```

PerfSyncBankTest (수업 21.01.26 nineteenth 내용)

```
class X extends Thread {
    public void run() {
        for(int i = 0; i < 1000000; i++) {
            // 현재 케이스는 실제 Critical Section에만 강제 동기화를 걸었다.
            // 그러므로 여러 태스크들이 동시에 접근할 수 있는 영역을
            // 부분적으로 안전하게 보호한 반면
            // 이전의 예제는 매서드 전체를 보호했다.
            // 그러므로 당연히 Critical Section만 방어할때에 비해 성능이 저하된다

            // PerfSyncBankTest 를 동기화 한다.
            synchronized (PerfSyncBankTest.psb) {
                // 동기화한 PerfSyncBankTest 의 plusMoney 3000을 할당.
                PerfSyncBankTest.psb.plusMoney(3000);
            }
        }

        // println은 왜 2개일까???
        // 실제 출력은 밑에 출력이 결과로 출력된다.
        // 하지만 위의 출력이 주석처리시 값이 변화된다.
        // 가정해보자면, 동기화로 인해 위의 출력을 선언해야지 값을
        // 불러올 수 있게되고, 그 후에 밑에 출력으로 출력한 것이라고 생각이든다.
        System.out.println(PerfSyncBankTest.psb.getPlusMsg());

        System.out.println("plusMoney(): " +
                PerfSyncBankTest.psb.getMoney());
    }
}

class Y extends Thread {
    public void run() {
        for(int i = 0; i < 1000000; i++) {
            synchronized (PerfSyncBankTest.psb) {
                PerfSyncBankTest.psb.minusMoney(1000);
            }
        }

        System.out.println(PerfSyncBankTest.psb.getMinusMsg());

        System.out.println("minusMoney(): " +
                PerfSyncBankTest.psb.getMoney());
    }
}

public class PerfSyncBankTest {
    public static PerfSyncBank psb = new PerfSyncBank();

    public static void main(String[] args) throws InterruptedException {
        System.out.println("원금: " + psb.getMoney());

        X x = new X();
        Y y = new Y();

        x.start();
        y.start();
    }
}
```

OperationAccelerator (수업 21.01.27 Twentieth 내용)

```java
public class OperationAccelerator {
    final int ONE = 1;

    private int dataStart;
    private int dataEnd;
    private int numOfData;
    private int maxThreadNum;

    public OperationAccelerator(int start, int end, int maxThreadNum) {
        // 원하는 숫자의 시작 숫자
        dataStart = start;
        // 원하는 숫자의 끝 숫자
        dataEnd = end;

        // 예를 들어보자. end : 10, start 1, One 1
        // 그러면 결과는 10 - 1 + 1 = 10
        // end : 100, start 1, One 1
        // 결과는 100 - 1 + 1 = 100
        // end : 100, start 5, One 1
        // 결과는 100 - 5 + 1 = 96
        // 이 공식을 사용하는 이유를 모르겠다.....
        numOfData = end - start + ONE;
        // ThreadNum 으로 몇개의 스레드를 사용할지 결정
        this.maxThreadNum = maxThreadNum;
    }
}
```

PerfSyncBankTest (수업 21.01.26 Twentieth 내용)

```java
public class AccelThread extends OperationAccelerator implements Runnable {
    private int localStart;
    private int localEnd;
    private int threadId;

    public AccelThread(int start, int end, int maxThreadNum, int id) {
        // 상속받은 OperationAccelerator 의
        // public OperationAccelerator(int start, int end, int maxThreadNum)
        // 가져온다.
        super(start, end, maxThreadNum);

        int total = end - start + 1;
        // 스레드에 정확히 나눠서 배분하기 위해서.
        // 예를들면 10000개의 데이터가 있다고 가정하자.
        // 그러면 스레드가 10개라면 1000개씩 검사할 것이다.
        // 그런데 만약에 10001개라면? 10개라면 1000개씩
        // 하고 나머지 한개를 누가 가져갈것인가????
        // 그러면 데이터가 삭제가 되거나 무시가 되면 데이터
        // 무결성원칙이 깨지기 때문에 아닐것 같으니
        // 랜덤으로 1개가 분배될 듯 하다.
        int threadPerData = total / maxThreadNum;

        localStart = id * threadPerData + 1;
        localEnd = localStart + threadPerData - 1;
        threadId = id;
    }

    @Override
    public void run() {
        System.out.printf("threadId = %d, localStart = %d\n", threadId, localStart);
        System.out.printf("threadId = %d, localEnd = %d\n", threadId, localEnd);
    }
}
```

PerformanceUtil (수업 21.01.26 Twentieth 내용)

```java
public class PerformanceUtil {
    // 시작시간
    public static long startTime;
    // 처리가 끝날때 까지 시간
    public static long estimatedTime;

    public static void performanceCheckStart() {
        // System.courrentTimeMillis() 는 현재시간을 구한다.
        // startTime 에 코드가 시작되고 난 현재 시간을 할당한다.
        startTime = System.currentTimeMillis();
    }

    public static void performanceCheckEnd() {
        // estimatedTime 에 코드가 끝난 현재 시간에 - 시작시간 의 계산값을 할당
        estimatedTime = System.currentTimeMillis() - startTime;
    }

    // 값을 가져오는 단위가 ms(1 / 1000 초)
    public static void printPerformance() {
        System.out.println(
                "걸린 시간: " + estimatedTime / 1000.0 + " s");
    }
}
```

PerformanceTest (수업 21.01.26 Twentieth 내용)

```java
public class PerfomancTest {
    public static void main(String[] args) {
        double sum = 0;

        // performanceCheckStart 시작간 할당
        PerformanceUtil.performanceCheckStart();

        // 10억번 반복
        for(int i = 1; i <= 2; i++) {
            // 이 공식이 잘 이해가 안됩니다
            // i x (거듭제곱 x i) x 삼각함수(i x 파이 / 180)
            sum += (i * (Math.pow(10, -15) * i)) * Math.sin(i * Math.PI / 180.0);
        }

        // performanceCheckEnd 끝나는 시 할당
        PerformanceUtil.performanceCheckEnd();

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

        // PerformanceUtil = 걸리는 시간 할당
        PerformanceUtil.printPerformance();
    }
}
```

PerformanceTest2 (수업 21.01.26 Twentieth 내용)

```java
public class PerformanceTest2 {
    final static int ZERO = 0;
    final static int END = 1000000000;
    final static int START = 1;
    // 10의 -15승을 구한다.
    final static double COEFFICIENT = Math.pow(10, -15);
    // 3.1416 라디안으로 변환시킨다.
    final static double DEG2RAD = 180.0;

    public static void main(String[] args) {
        double sum = ZERO;

        // 코드를 시작한 현재시간을 할당
        PerformanceUtil.performanceCheckStart();

        for(int i = START; i <= END; i++) {
            sum += (i * (COEFFICIENT * i)) * Math.sin(i * Math.PI / DEG2RAD);
        }

        // 현재 코드의 끝난 시간 - 시작시간 의 계산값을 할당
        PerformanceUtil.performanceCheckEnd();

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

        // 총 걸린시간을 할당
        PerformanceUtil.printPerformance();
    }
}
```