스프링 스터디

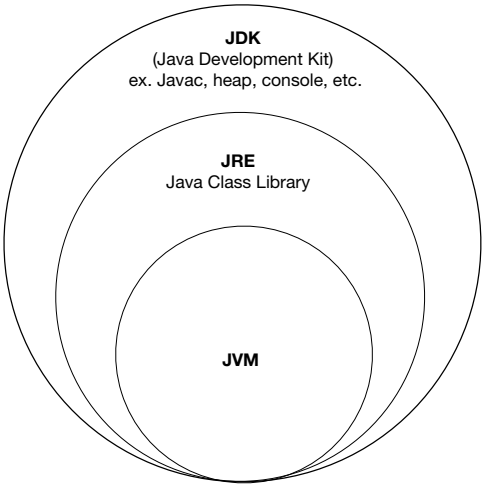
2장. 자바와 절차적/구조적 프로그래밍

(1) 자바 프로그램의 개발과 구동

■ JVM [Java Virtual Machine]: 자바를 구동하기 위한 가상 머신
JDK(자바 개발 도구)로 개발된 프로그램은 JRE(자바 실행 환경)에 의해 JVM(가상의 컴퓨터)에서 구동됨

JDK는 JRE를 포함하며, JRE는 JVM을 포함하는 형태로 배포됨

자바는 JVM 덕분에 운영체제에 종속되지 않고 실행되나, JVM은 운영체제 별로 필요함



JDK : Java Development Kit
⇒ 자바 개발 도구
JDK = JRE + 개발을 위한 도구
ex. 컴파일러, 디버그 도구 등
javac.exe

버전과 종류가 존재함

JRE : Java Runtime Environment
⇒ 자바 실행환경
JRE = JVM + 자바 프로그램에 필요한 라이브러리 파일 등
JVM의 실행환경 구현

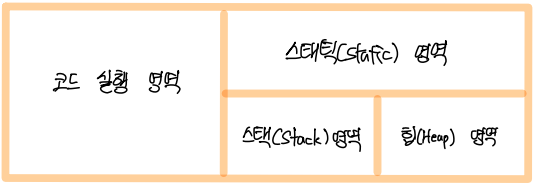
JVM : Java Virtual Machine
OS 별로 존재함
바이트코드를 읽고 검증하고 실행함

JDK로 개발된 프로그램은 JRE에 의해 JVM에서 구동

☆ Java의 버전 = JDK 버전

Write Once Run Anywhere
과거 각 운영체제에 맞게 파일을 준비해야 하였으나, 자바는 이를 JVM으로 해결함
자바 개발자는 운영체제에 설치된 JVM용으로 프로그램을 작성하고 배포하면
각 플랫폼에 맞는 JVM이 중재자가 되어 플랫폼에서 프로그램을 구동하는 데 문제가 없게 만들어줌

■ 프로그램 메모리 사용 방식 [T 메모리 구조]



→ 객체지향 프로그래밍 메모리 사용 방식

■ 자바에 존재하는 절차적/구조적 프로그래밍 유산

객체지향 프로그래밍은 절차적/구조적 프로그래밍으로 부터 만들어짐

절차지향 프로그래밍을 요약하자면 goto를 사용하지 말라는 것이며, 자바에서는 이를 막고자 goto를 예약어로 등록하였음

abstract	continue	for	New	switch
assert	default	goto	Package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw

검은색 부분은 '절차 -> 객체' 넘어온 예약어들임 (이 외에도 더 존재함)

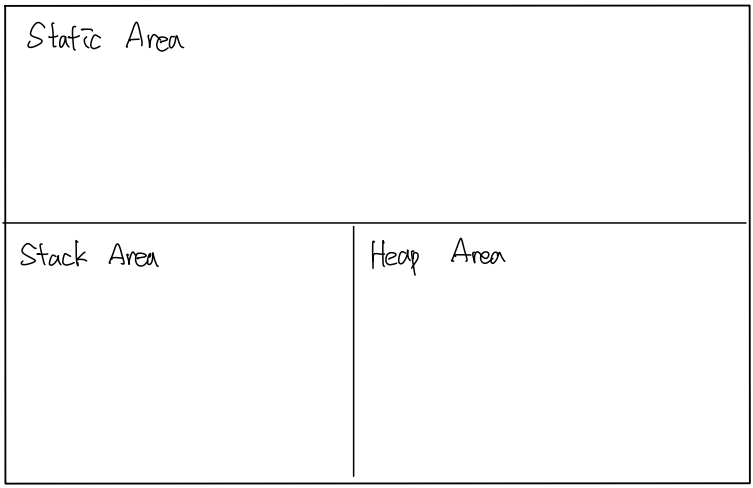
goto를 막은 이유
프로그램의 실행 순서가 이리저리 이동이 가능해지는데 프로그램 논리적 구성을 잘 설계한다면 이는 피할 수 있는 것임

메서드에서도 절차적/구조적 프로그래밍의 유산 확인 가능

함수는 중복 코드 제거와 논리를 분할하기 위한 용도로, 객체지향 언어에서 메서드와 같은 것임
그리고 객체지향 언어에서 제어문이 존재할 수 있는 유일한 공간은 메서드 내부임

함수와 메서드의 차이? 없음. 굳이 따지자면, 메서드는 반드시 클래스 정의 안에 존재해야 하지만, 함수는 클래스나 객체와 아무 관계가 없음

■ main() 메서드 : 메서드 스택 프레임



Static → Class Area

Stack → Method Area

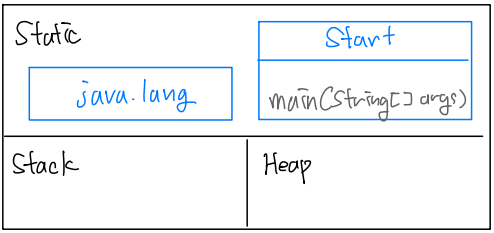
Heap → Object Area

■ 예제 2-1 : Start.java

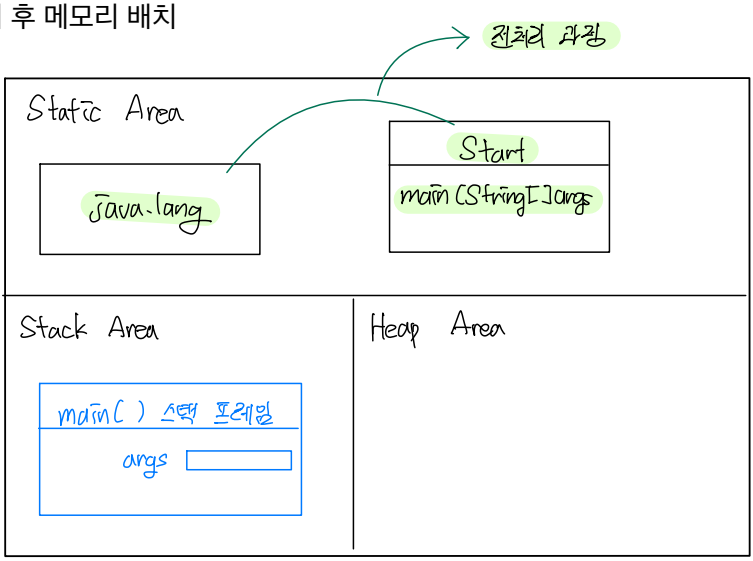
```
1  public class Start {
2    public static void main(String[] args) {
3      System.out.println("Hello OOP!!!");
4    }
5  }
```

실행 순서

(1) JRE : 프로그램에 'main( )' 메서드가 존재하는지 확인
(2) JVM의 실행을 준비해, JVM은 목적 파일을 받아 실행
(3) JVM의 전처리 과정 진행
  (a) java.lang 패키지를 스태틱 영역에 배치
  (b) 임포트 패키지를 스태틱 영역에 배치
  (c) 프로그램 상 모든 클래스를 스태틱 영역에 배치



■ 전처리 후 메모리 배치



중괄호 ({ })를 만날 때 마다 스택 프레임에 하나씩 생성됨

닫는 중괄호를 만나면, 스택 프레임이 소멸됨

main( ) method는 자바 프로그램의 시작이자 끝 지점임

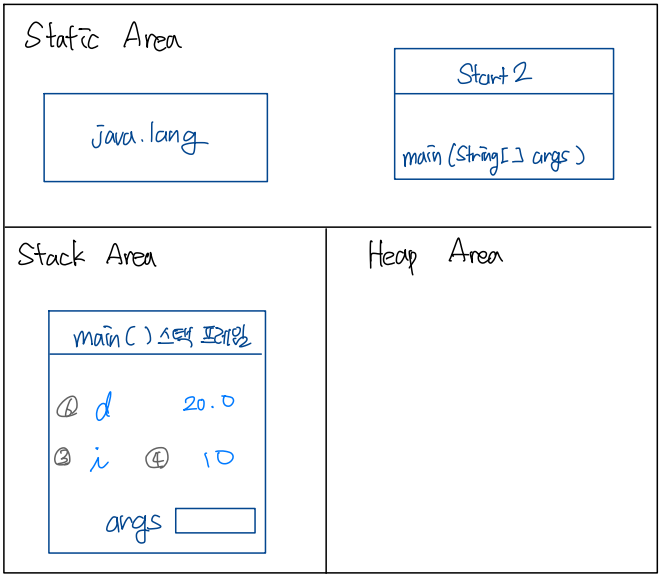
main( ) method 종료 시 JRE는 JVM을 종료하고, JRE 자체도 운영체제 메모리에서 사라짐

→ ① Memory 소멸
② JVM 가동 종료
③ JRE 사용 시스템 자원 반납

■ 메모리와 변수

■ 예제 2-2 : Start2.java

```
1  public class Start2 {
2    public static void main(String[] args) {
3      int i;
4      i = 10;
5
6      double d = 20.0;
7    }
8  }
```



■ 블록 구문과 메모리: 블록 스택 프레임

```
1  public class Start3 {
2    public static void main(String[] args) {
3      int i = 10;
4      int k = 20;
5
6      if(i == 10) {
7        int m = k + 5;
8          k = m;
9      } else {
10       int p = k + 10;
11         k = p;
12     }
13
14     //k = m + p;
15   }
16 }
```

→ 각 스택 프레임은 종료 중괄호에서 소멸됨.

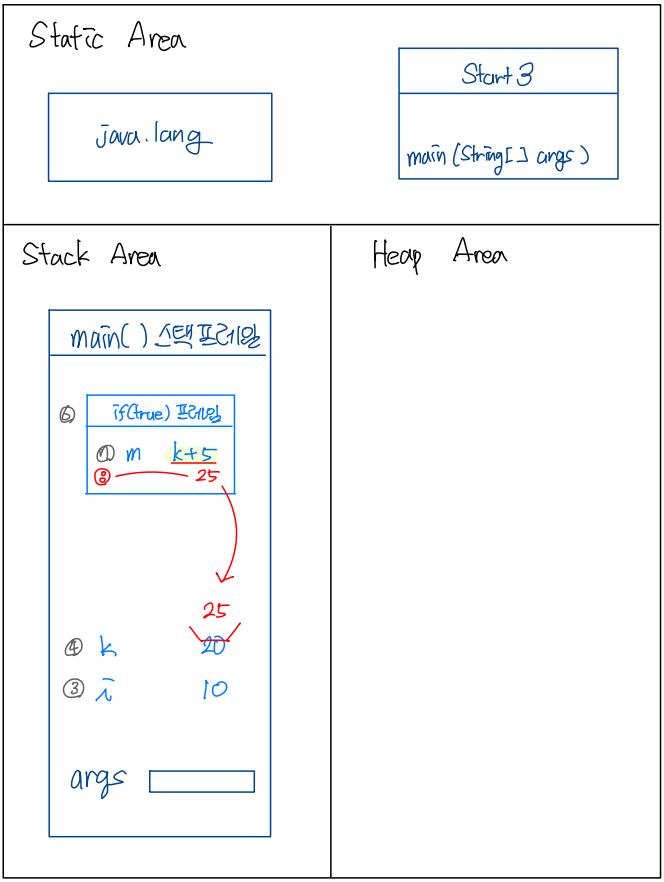
//k = m + p; ⇒ 그렇게에 main( ) 스택 프레임에 m은 존재하지 않는 것이므로 에러가 발생하는 것임. (컴파일 에러 발생)

p can't be resolved to a problem.

■ 예제 2-3 : Start3.java

외부 스택 프레임에서 내부 스택 프레임의 변수에 접근하는 것은 불가능하나 그 역은 가능하다.

→ main( ) stack frame은 if(true) stack frame의 m 변수에 접근할 수 없지만 if(true) stack frame은 main( ) stack frum의 k 변수에 접근 할 수 있음.



■ 변수와 메모리

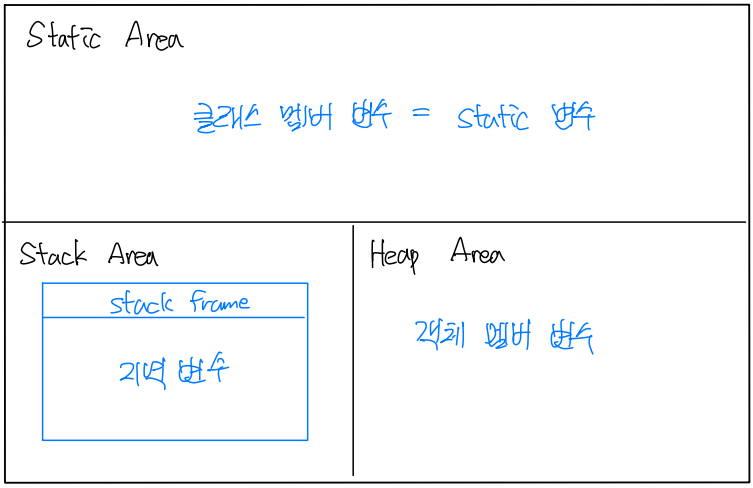
- 변수는 메모리에 존재하며, 선언된 변수의 특성에 따라 위치가 달라짐.

- 지역 변수, 클래스 멤버 변수, 객체 멤버 변수

- 지역 변수 : Stack Frame
  스택 프레임 안에서 일생을 보내며, 스택 프레임과 함께 소멸됨

- 클래스 멤버 변수 : Static Area
  스태틱 영역에 존재하며, JVM이 종료될 때까지 고정(static) 상태로 자리를 지킴

- 객체 멤버 변수 : Heap Area
  객체와 함께 가비지 컬렉터라는 힙 메모리 회수기에 의해 일생을 마침



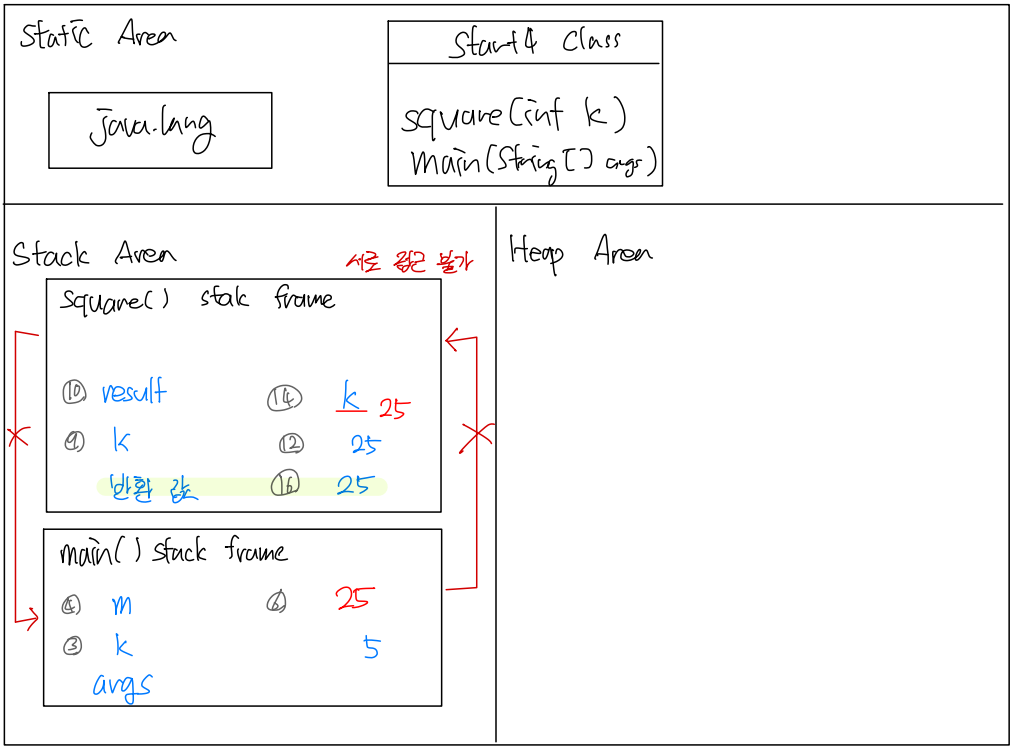
■ 메서드 호출과 메모리 : 메서드 스택 프레임

```
1  public class Start4 {
2    public static void main(String[] args) {
3      int k = 5;
4      int m;
5
6      m = square(k);
7    }
8
9    private static int square(int k) {
10     int result;
11
12     k = 25;
13
14     result = k;
15
16     return result;
17   }
18 }
```

■ 예제 2-5 Start4.java

Call by value (값에 의한 호출)
→ 변수가 저장하는 값만 복제해서 전달

"메서드 블랙박스화"
→ 메서드 사이 입력 값과 반환 값으로만
메서드 사이의 값이 저장될 뿐
서로 내부 지역변수는 볼 수 없음.



■ 메서드 블랙박스화 이유

- 포인터 문제 때문. Main 메서드에서 square 메서드의 result 변수에 접근하려면 메모리의 위치인 메모리 주소 값을 알아야 하는데, 자바는 이 포인터가 없기에 불가

- 즉, 자바는 포인터가 존재하지 않아 메서드 스택 프레임 사이에 변수를 참조하는 것은 불가함

■ 메서드 사이 값 전달 방법

- 매개변수, 반환 값(return 값), 전역변수

■ 예제 2-6. Start5.java

```
1  public class Start5 {
2    static int share;
3
4    public static void main(String[] args) {
5      share = 55;
6
7      int k = fun(5, 7);
8
9      System.out.println(share);
10   }
11
12   private static int fun(int m, int p) {
13     share = m + p;
14
15     return m - p;
16   }
17 }
```

* share

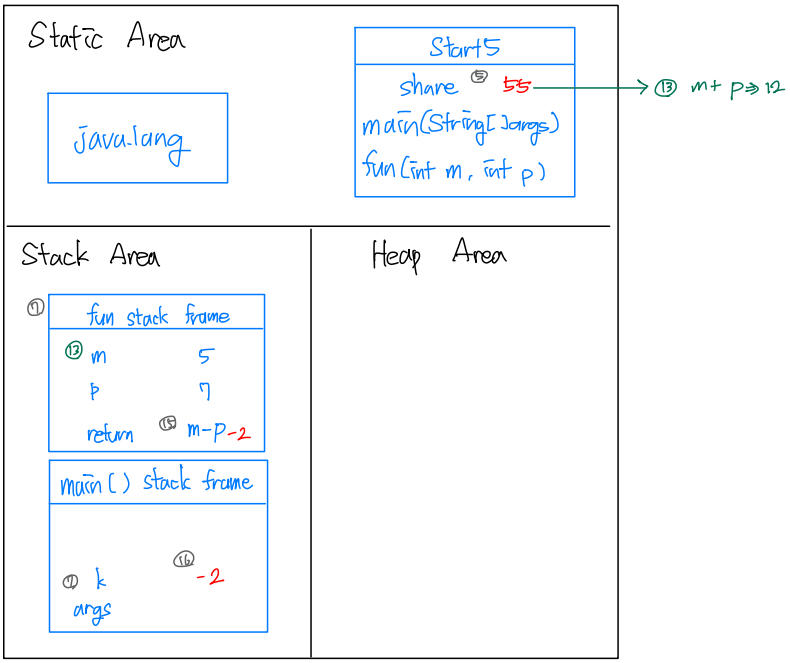
→ 메모리에서 모든 메서드 사이에서 공유해서 사용할 수 있음.

* 지역변수 vs 전역변수

- 지역변수 : 스택 프레임에 종속적
- 전역변수 : 스택 프레임에 독립적

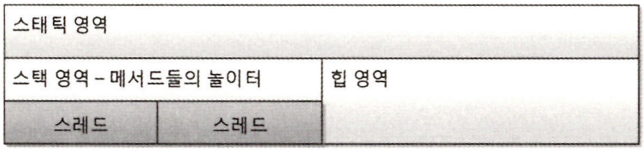
* 전역변수

→ 유지보수가 어려워질 수 있어 '전역상수변수'를 사용하는 것 외에 비 권장


Static Area
java.lang
Start5
share ⑤ 55
main(String[] args)
fun(int m, int p)
⑬ m+p⇒12
Stack Area
Heap Area
① fun stack frame
⑬ m 5
p 7
return ⑮ m-p-2
main() stack frame
⑯ -2
① k
args

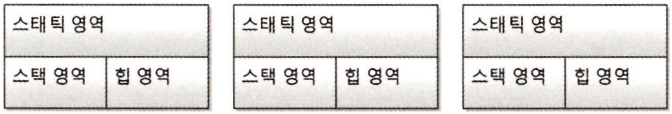
■ 멀티 스레드 / 멀티 프로세스의 이해

- 멀티 스레드 (Multi Thread)
  T 메모리 모델의 스택 영역을 스레드 개수만큼 분할해 사용



[그림 2-40] 멀티 스레드는 스택 영역을 스레드 개수만큼 분할해서 사용

- 멀티 프로세스 (Multi Process)
  다수의 데이터 저장 영역으로 다수의 T 메모리를 갖는 구조



[그림 2-41] 멀티 프로세스는 자료 저장 영역에 다수의 T 메모리를 사용

■ 멀티 스레드 VS 멀티 프로세스

멀티 프로세스는 각 프로세스마다 각자의 T 메모리가 존재하며, 각자 고유 공간이므로 서로 참조가 불가하며, 메모리 사용량이 큼

멀티 스레드는 하나의 T 메모리 안에서 스택 영역만 분할한 것이기에 하나의 스레드에서 다른 스레드의 스택 영역에 접근할 수 없지만, 스태틱 영역과 힙 영역은 공유해 사용함 고로 멀티 프로세스 대비 메모리 사용량이 적음

+ 서블릿(Servlet)은 요청당 스레드를 생성함

■ 멀티 스레드에서의 전역변수

각 스레드에서 전역변수에 서로 다른 값을 할당해 이를 출력할 경우, CPU의 스케쥴러에 의해 실행되는 순서에 따라 의도한 값과 달리 할당되는 문제가 발생함

ex. A 스레드 -> 10 할당, B 스레드 -> 20 할당. A에서 출력 시도. 입력 값과 다름.

이를 '스레드의 안전성이 깨진다.'라고 표현함.

■ 예제 2-7. Start6.java

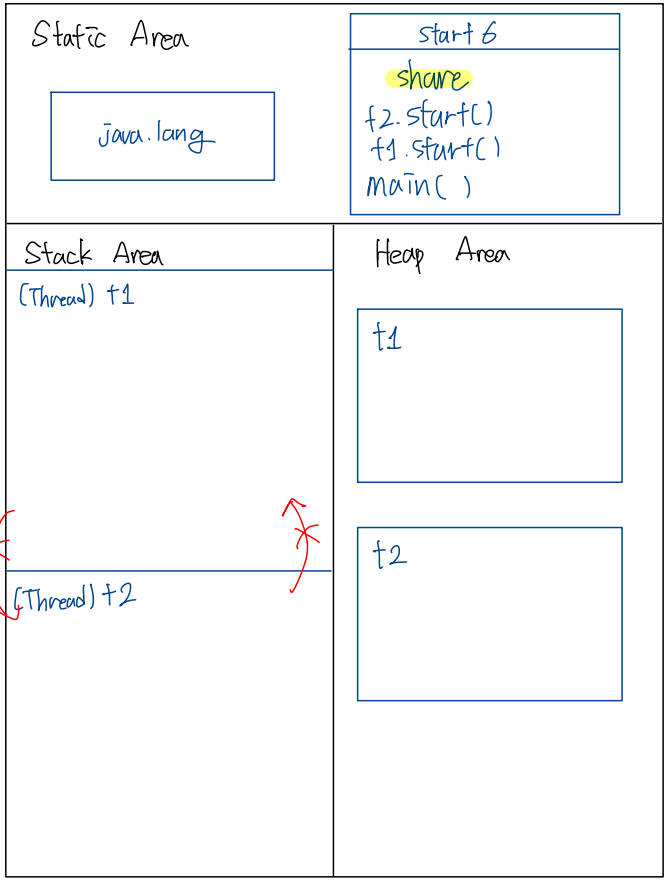
```java
// 예제 2-7. 71페이지
public class Start6 implements BackJoon {

    static int share;

    class Start6Class extends Thread {

        public void run(){

            for(int i=0; i < 10; i++)
                System.out.printf("multi thread %d: %d\n", i, share++);

            try {
                sleep( millis: 1000);    // imported from java.lang.Thread.sleep
            } catch (InterruptedException e){

            }

        }

    }

    @Override
    public void start() {

        Start6Class s1 = new Start6Class();
        Start6Class s2 = new Start6Class();

        s1.run();
        System.out.println("---------------------------------");
        s2.run();

    }
```

→ Start6Class가 생성되며 Thread가 생성됨

```
multi thread 0: 0
multi thread 1: 1
multi thread 2: 2
multi thread 3: 3
multi thread 4: 4
multi thread 5: 5
multi thread 6: 6
multi thread 7: 7
multi thread 8: 8
multi thread 9: 9
---------------------------------
multi thread 0: 10
multi thread 1: 11
multi thread 2: 12
multi thread 3: 13
multi thread 4: 14
multi thread 5: 15
multi thread 6: 16
multi thread 7: 17
multi thread 8: 18
multi thread 9: 19
```

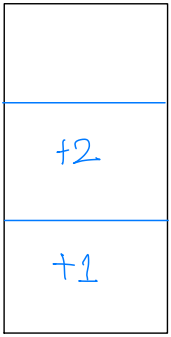
객체가 생성되며 Heap 영역에서 참조하는 각자의 주소를 갖게 됨. 하지만 Muti Thread는 stack을 제외한 나머지 영역을 공유하기에 왼쪽 결과가 나타나는 것임.

이를 T 메모리로 그리자면 다음과 같음.



→ 예제 코드 기준의 T메모리 임.

CPU에는 순서상 (stack = FIFO)



↳ t1.run()을 먼저 실행.
System.out.println(share++);
이로 share값 증가.

↳ Thread간 접근은 불가하지만 static 영역은 공유되기에 CPU상 다음 실행 구간인 t2.run() 실행시 증가되어 있던 다음 값부터 출력 및 증가 실행.