2021.06.02 Java

```java
public class _99th_ArrayPractice {
    public static void main(String[] args) {

        // 이중배열 선언
        int[][] int_arr;
        int[] int_arr2[];
        int int_arr3[][];

        // 이중배열 초기화
        int_arr = new int[2][3];

        int_arr2 = new int[][] {{1,2,3,4,5}, {1,3,4,5,9}};
                // int_arr2 = new int[2][5]에 값이 넣어진 것.

        double[][] arr_double = new double[2][3];
        System.out.println(arr_double.length); // arr_double의 length는 2
        for (int i = 0; i < arr_double.length; i++){
            for (int j = 1; j < arr_double[i].length + 1; j++){
                arr_double[i][j - 1] = j;
                System.out.print(arr_double[i][j - 1] + " ");
            }
            System.out.println();
        }
```

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

'int_arr = new int[2][3]"을 도식화한 것

```
"C:\Program Files\Java\jdk-16\bi
2
1.0 2.0 3.0
1.0 2.0 3.0
0.0 1.0 2.0 3.0 4.0 5.0 6.0
0.0 1.0 2.0 3.0 4.0 5.0 6.0
0.0
```

〈이중배열〉

이중배열 - 다차원배열(다중배열)

이중배열을 이해하면 삼중배열 / 사중배열도 이해할 수 있지만 코드가 너무 복잡해지기 때문에 사용x

단일배열이 하나의 주소값을 참조하여 그 주소값부터 배열의 크기만큼 일렬로 나열된 것이라면

이중배열은 여러개의 주소를 참조하여 여러열을 나열시킨 것.

int_arr = new int[3][5]; 는

5개의 열이 3줄 있다는 뜻.

```
int_arr3 = new int[3][]; // 동적배열...?
// 3개의 줄을 세울건데 몇 개씩 어떤 값이 들어갈지 정해져있지 않다
int_arr3.add(); // 근데 왜 add 안되는거지..?
```

```
double[][] arr_double_2 = new double[3][];
for (int i = 0; i < arr_double_2.length; i++){
    int rand = (int) (Math.random() * 10 + 1);
    arr_double_2[i]=new double[rand]; // arr_double_2[i]를 double[rand]만큼 크기를 준다.
    for (int j = 0; j < arr_double_2[i].length; j++){
        arr_double_2[i][j] = j;
        System.out.print(arr_double_2[i][j] + " ");
    }
    System.out.println();
}
```

```
"C:\Program Files\Java\jdk-16\bi
2
1.0 2.0 3.0
1.0 2.0 3.0
0.0 1.0 2.0 3.0 4.0 5.0 6.0
0.0 1.0 2.0 3.0 4.0 5.0 6.0
0.0
```

```
double[][] arr_double = new double[2][3];
System.out.println(arr_double.length); // arr_double의 length는 2
for (int i = 0; i < arr_double.length; i++){ //arr_double[i]의 length는 3
    for (int j = 0; j < arr_double[i].length; j++){
        arr_double[i][j] = j;
        System.out.print(arr_double[i][j] + " ");
    }
    System.out.println();
}
```

```
2
0.0 1.0 2.0
0.0 1.0 2.0
0.0 1.0 2.0 3.0
0.0 1.0 2.0 3.0 4.0 5.0
0.0 1.0 2.0 3.0 4.0 5.0
```

위에 arr_double 구하는 식에서 나중에 헷갈릴까봐 그냥 j=0부터 시작하고 식에서 -1들 뺌

〈배열에 관한 기본 개념〉

int 배열의 default 값은 0 ›› 초기화 할 때 index수만 초기화 하면 값들은 다 0으로 setting

short,int,double,float형의 default값은 0

String 형의 default값은 null

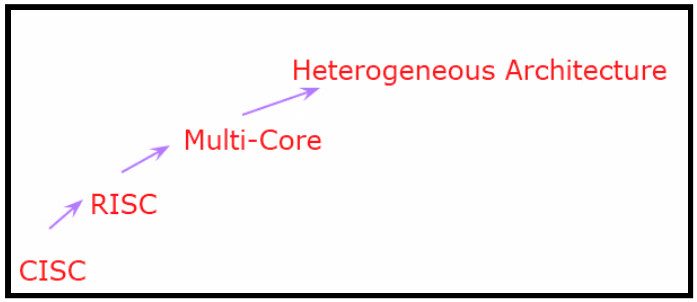
char 형의 default 값은 공백

〈Thread 개념정리〉

- CPU는 오직 한 순간에 한 가지 일만 할 수 있다.

- Thread는 CPU 개수만큼 1:1로 할당 가능하다. Thread : Process의 실행 가능한 최소 단위

-



CPU의 발전사

실제로 Thread가 개입된 부분은 multi-core부터.

- 동기화(Synchronization): 한 cpu(thread)에서 어떤 프로세스를 처리하는 동안 다른 cpu(thread)는 그와 동일한 프로세스에는 관여하지 말 것 ›› 하나의 자원을 여러 태스크가 사용하려 할 때 한 시점에서 하나의 태스크만이 사용할 수 있도록 하는 것이다.

  데이터의 무결성 확보에 매우 중요

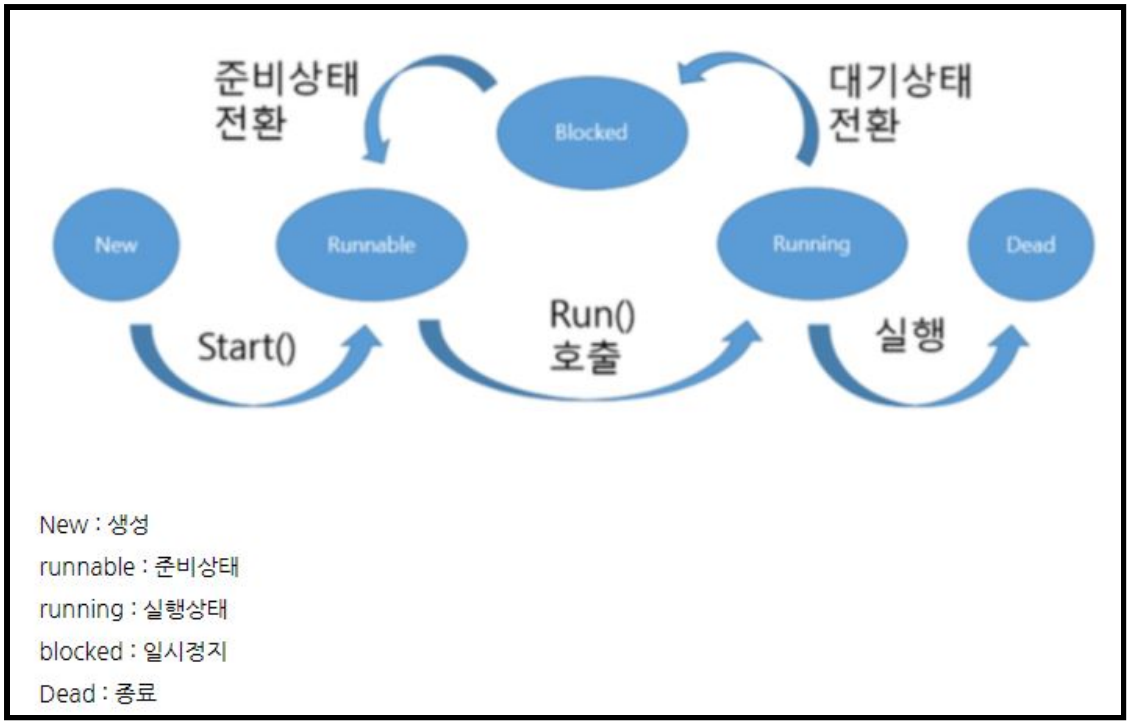
  이런 Synchronization Mechanism에는 Mutex/Spinlock/Semaphore 등이 있음.

  (Mutex는 context-switching 사용 / Spinlock은 해당 프로세스가 끝날 때까지 다른 작업 못 하게 lock)

- Program: 실행가능한 소스코드 집합 / Process: 메모리를 할당 받아 실행중인 프로그램 / CPU의 추상화

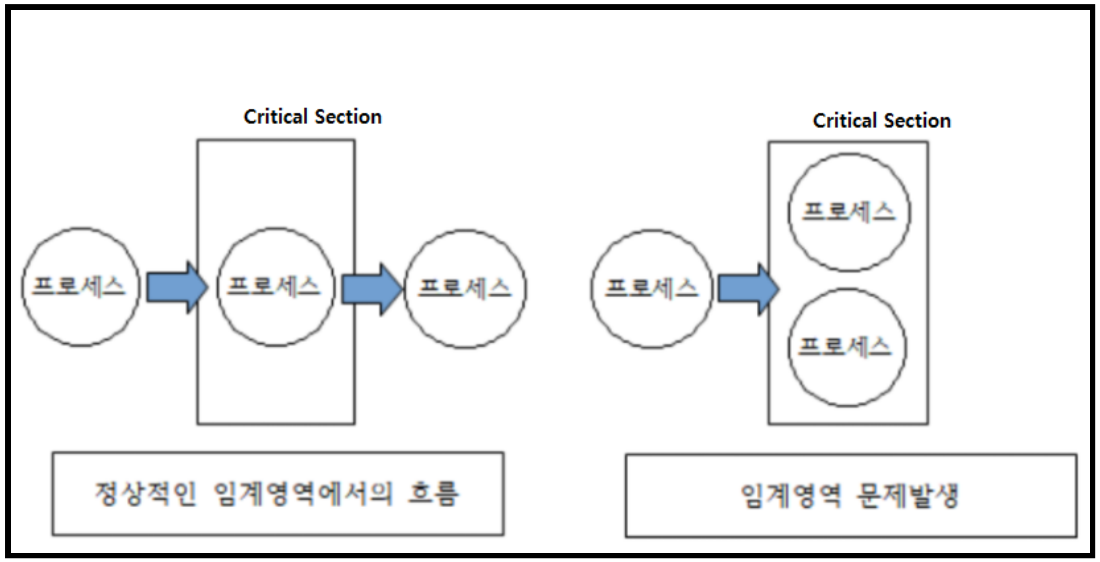
  (process = (memory+data)+thread) / Processor: CPU

Thread의 생명주기



New : 생성
runnable : 준비상태
running : 실행상태
blocked : 일시정지
Dead : 종료

- **Critical Section(임계영역)** 다수의 프로세스가 접근 가능한 영역이면서도 한 번에 하나의 프로세스만 사용할 수 있는 영역.

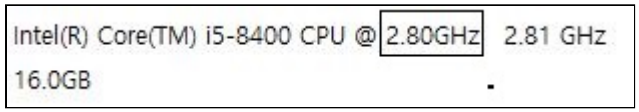


위에 기재한 Mutex/Spinlock/Semaphore 등이 Critical Section에 문제가 발생하지 않도록 방어.
mutex/spinlock/semaphore... 는 프로그램의 개념은 아님.
(차후에 더 알아볼것)

여러개의 thread가 동시다발적으로 접근할 수 있는 데이터는 전부 Critical Section.

전역변수이더라도 그 데이터를 사용하는 thread가 하나 뿐이라면 Critical Section(x)

- Multi-Tasking > 아주 빠르게 순차적으로 동작하는 것. 너무 빠르기 때문에 동시에 발생하는 것 처럼 느껴지는 것 뿐.



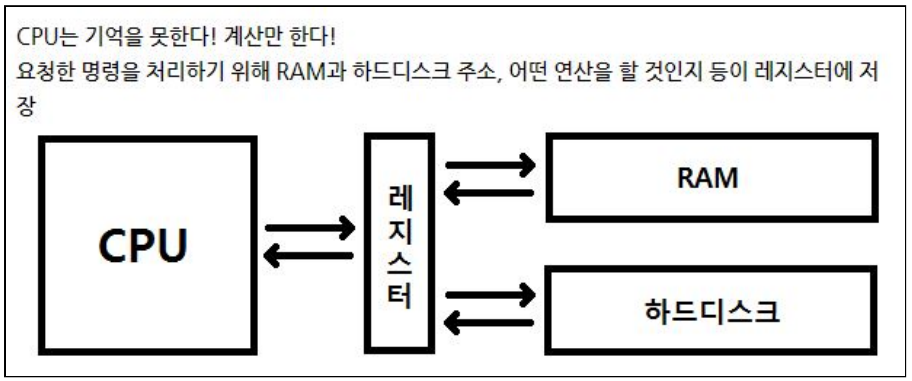
Hz = 초당 진동수 = [컴퓨터] 클럭(clock) = 초당 몇개의 명령어를 처리 할 수 있는가
G = 10^9 = 10억 / 2.80Ghz: 1초당 28억개의 명령어 처리 가능
multi-tasking에는 context switching이라는 개념이 들어가있음.

- **Context-Switching**

CPU의 사용 권한 자체는 O/S가 적절하게 스케줄링해서 권한 부여 + 자원 부여(memory + data)

Context: CPU가 해당 프로세스를 실행하기 위한 해당 프로세스의 정보들(레지스터값)

Context-Switching: 멀티프로세스 환경에서 CPU가 하나의 프로세스를 실행하고 있는 상태에서 인터럽트 요청에 의해 다음 우선 순위의 프로세스가 실행되어야 할 때, 기존의 프로세스의 상태 또는 레지스터값(context)을 main Memory에 저장하고 CPU가 다음 프로세스를 수행하도록 새로운 프로세스의 상태 또는 레지스터값을 교체하는 작업.



하드디스크 - 보조 기억장치(영구저장공간)

RAM - 주 기억장치(임시저장공간)

레지스터 - CPU가 요청을 처리하는데 필요한 데이터를 일시적으로 저장하는 기억장치
메모리와 하드디스크에 연산결과를 보내는 등의 명령을 처리하기 위해
메모리와 하드디스크의 주소와 명령의 종류를 저장하는 공간. CPU는 자체적으로 데이터들을 저장할 방법이 없기 때문에 메모리로 직접 데이터를 전송할 수 없다.
때문에 CPU연산을 위해서는 반드시 레지스터를 거친다.

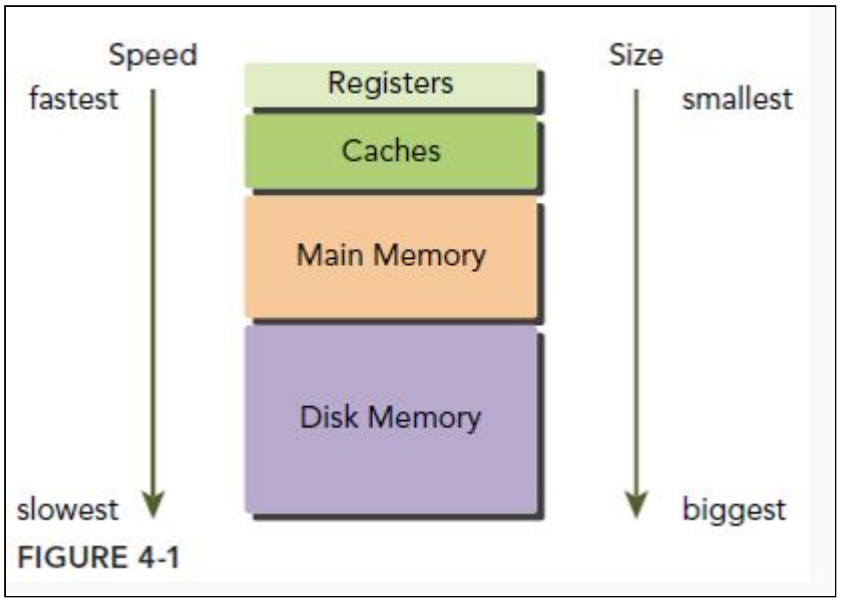


FIGURE 4-1

```java
import java.util.Random;
class Car implements Runnable{
                //Runnable interface 사용.
                //즉, Runnable에 미구현된 run() method를 여기서 구현해야한다.
    String name;
    private int sleepTime;
    private final static Random generator = new Random(); // Random class의 객체 generator 생성

    public Car(String name){
        this.name = name;
        //Random 클래스로 만든 객체에 nextInt() 매서드를 통해서도 랜덤값을 생성할 수 있다.
        sleepTime = generator.nextInt( bound: 3000) + 3000;
    }
    // thread를 사용할 때에는 무조건 run() method를 구현해야됨. 매우중요
     @Override
     public void run() {
         // try라는 키워드를 적는 경우는 I/O나 특정한 이벤트,
         // 인터럽트 등등이 발생하는 경우에 작성하게 됨
         // "이 코드는 에러를 유발할 수 있다"를 암시함
        try {
            Thread.sleep(sleepTime);
            // 정말 에러가 발생했다면 catch 실행.
            // 물론 Thread.sleep()에서 에러가 발생할 일은 99.999999999999%로 없다.
        } catch(InterruptedException e) {
            System.out.println("error가 나면 이 문장이 print된다.");
        }
        System.out.println(name + "이 경주를 완료하였습니다.");
        //thread.sleep method에서 난수로 지정한 sleepTime이 가장 낮게 나온 차가
        // 제일 경주를 빨리 완료함.
     }
```

```java
@FunctionalInterface
public interface Runnable {
    When an object implementing interface Runna
    causes the object's run method to be called ir
    The general contract of the method run is tha
    See Also: Thread.run()
    public abstract void run();
```

〈Thread 사용의 기본예제〉

```java
public class _3rd_ThreadEdu {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Car( name: "Ferrrari"));
        Thread t2 = new Thread(new Car( name: "Spyder 918"));
        Thread t3 = new Thread(new Car( name: "Maserati MC20"));

        t1.start(); // Thread class의 method 중 하나.
        t2.start();
        t3.start();
```

```
Spyder 918이 경주를 완료하였습니다.
Maserati MC20이 경주를 완료하였습니다.
Ferrrari이 경주를 완료하였습니다.
```

```
Ferrrari이 경주를 완료하였습니다.
Maserati MC20이 경주를 완료하였습니다.
Spyder 918이 경주를 완료하였습니다.
```

```java
public class Thread implements Runnable {
    /* Make sure registerNatives is the first th
    private static native void registerNatives()
    static {
        registerNatives();
```

쓰레드를 구현하는 방법은 크게 두 가지가 있다.

1. Thread클래스를 상속받는 방법.
2. Runnable인터페이스를 구현하는 방법이다.

>> 일반적으로 Runnable 인터페이스를 구현한다. (Class Thread 자체도 이미 Runnable interface를 Implements하고 있다.)

그 이유는.

1) Thread클래스를 상속받으면 다른 클래스를 상속받을 수 없다.
2) 재사용성이 높다.
3) 코드의 일관성을 유지할 수 있다.
4) 보다 객체지향적인 방법이다.

두 방법 중 어떤 것을 사용하더라도

run() 이라는 추상메소드의 몸통을 정의해주어야 한다.

〈Start() method〉

main method에서 run()을 호출하는 것은 생성된 thread를 실행시키는 것이 아니라 단순히 class에 선언된 method를 호출하는 것일 뿐.

반면 start()는 새로운 thread가 작업을 실행하는데 필요한 호출스택(call stack)을 생성한 다음 run()을 호출한다.

즉, start()는 Thread를 실행시키기 위한 method. But method를 호출하였다고 해서 thread가 바로 실행되는 것은 아니다.

thread는 OS가 작성한 스케줄에 따라 그 실행순서와 실행시간이 정해지는데 start() method를 호출하였더라도 순서가 아니라면
실행대기 상태에 있다가 자기차례가 되면 실행된다. 그리고 주어진 실행시간 동안 작업을 마치지 못한 thread도 다시 자신의 차례가 돌아올 때까지 대기상태.

따라서 이 method를 다시 정리하자면,

'start() method를 호출한 thread가 실행순서가 됐을 때 thread를 실행하라 '라는 뜻.

또 start() 메소드는 단 한 번만 호출될 수 있다는 특징이 있다. 다시 말해, 한번 종료된 thread는 다시 실행될 수 없다는 것이다.

= 작업을 마친 thread는 call stack이 모두 비워지면서 사라짐.

start() 메소드를 두 번 실행하고자 한다면 새로운 thread를 다시 생성하여 method를 호출해야 된다.

main method가 수행을 마쳤더라도 다른 thread가 아직 작업을 마치지 않은 상태라면 프로그램은 종료되지 않는다.

》 실행중인 사용자 thread가 하나도 없을 때 프로그램이 종료됨.

- 한 thread가 예외가 발생해서 종료되어도 다른 thread의 실행에는 영향을 미치지 않는다.