

# 디지털컨버전스 기반 UI/UX Front 전문 개발자양성 과정(비트캠프)

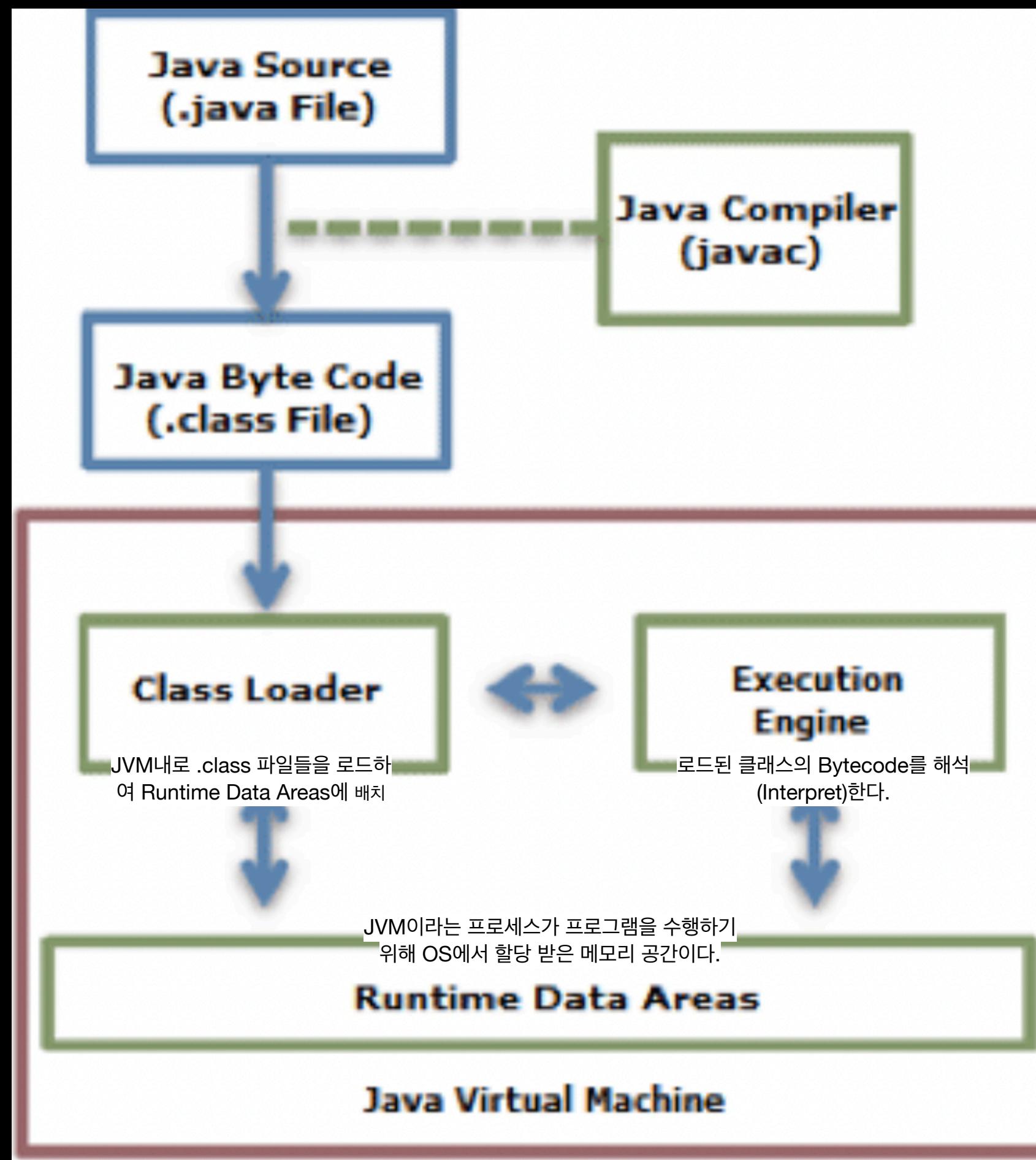
강사: 이승훈

수강생: 오진욱

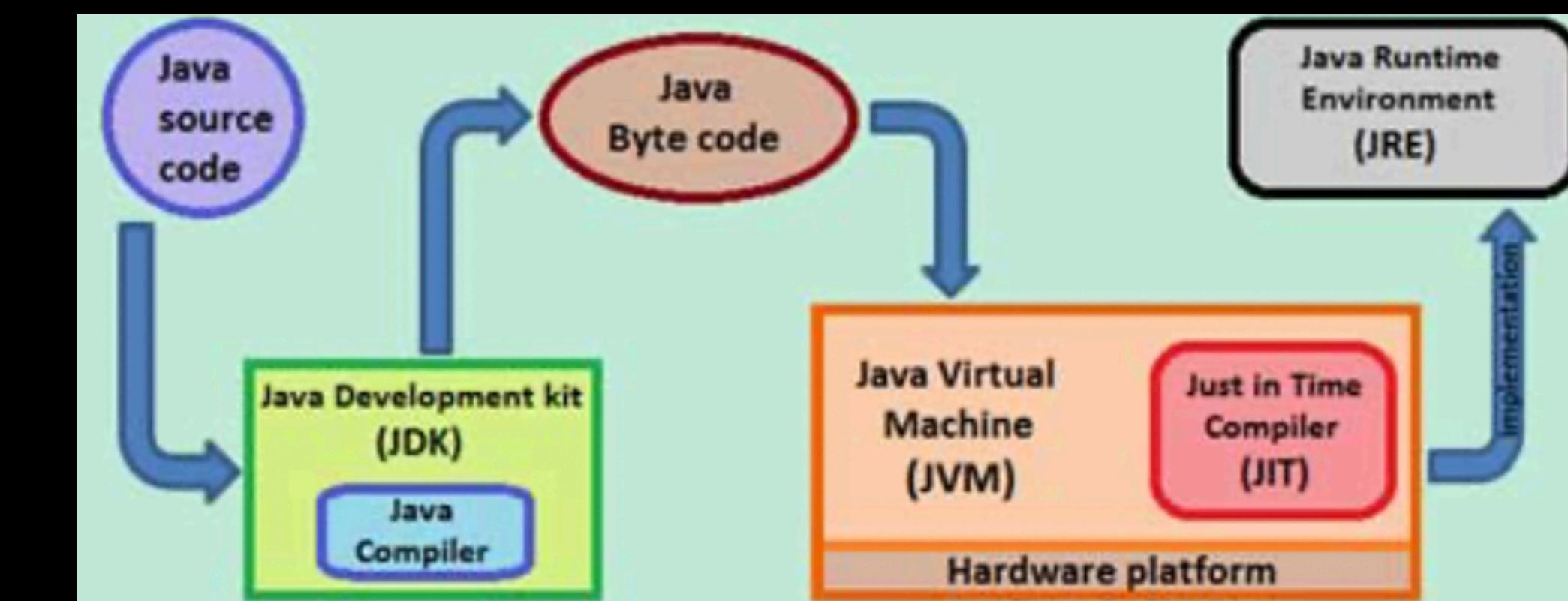
- Java -

# CS/OS

## Java Virtual Machine

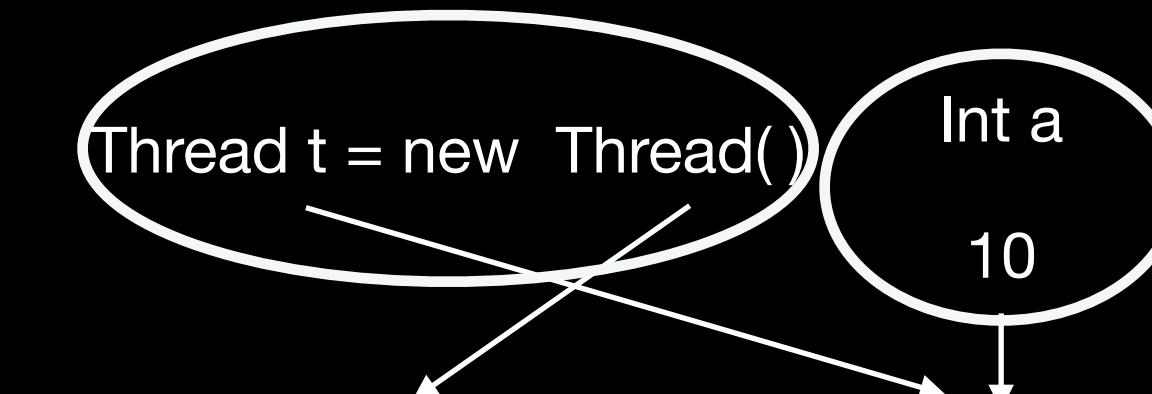
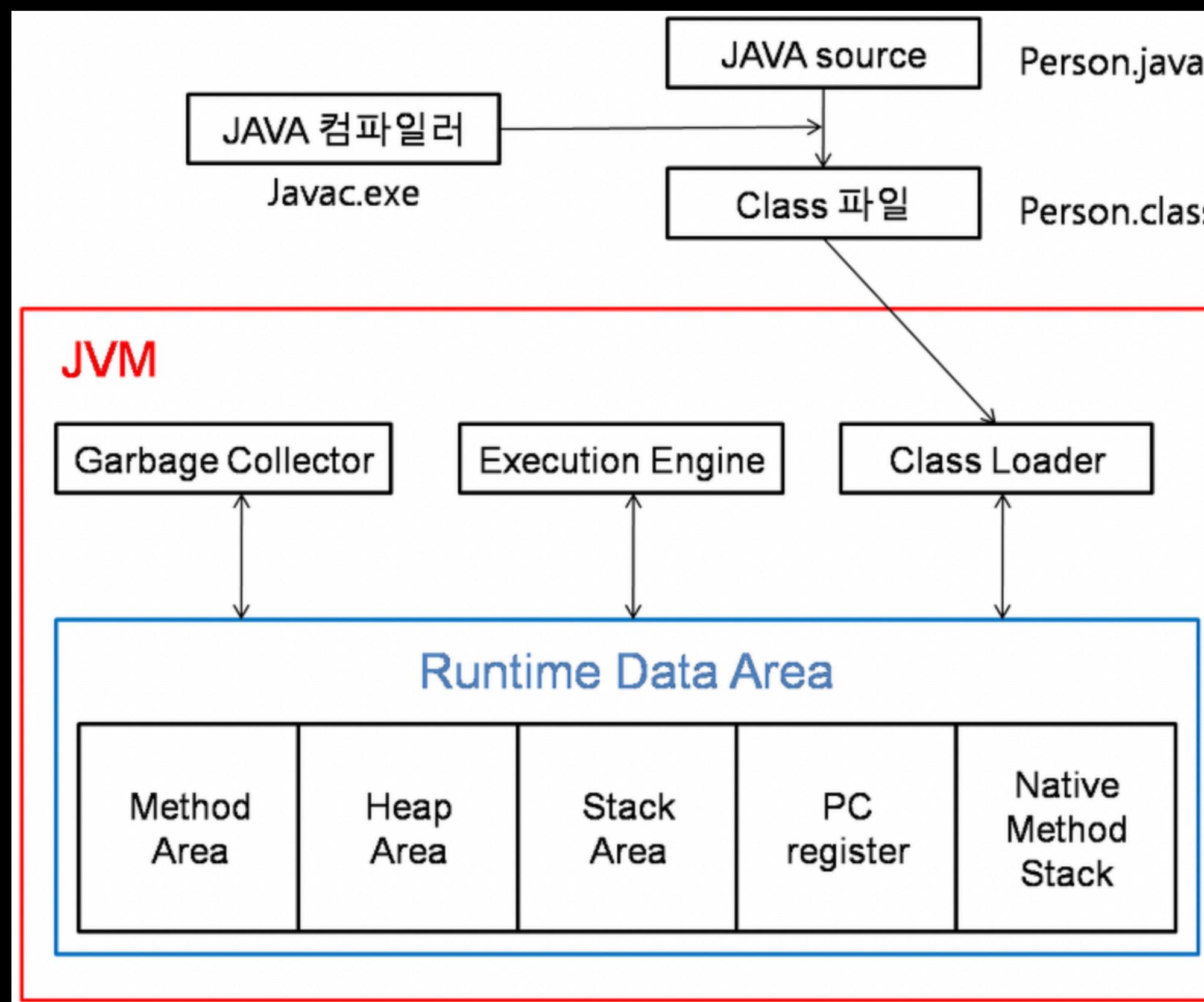


- 자바의 가상머신 -> 자바 바이트 코드를 실행 할 수 있는 주체
- 운영체제 위에서 동작하는 프로세스로 자바 코드를 컴파일해서 얻은 바이트 코드를 해당 운영체제가 이해할 수 있는 기계어로 바꿔 실행 시켜주는 역할



# CS/OS

## Java Virtual Machine



- Method Area(Data)
  - Heap Area
  - Stack Area
- 필드 정보
  - new 키워드로 생성된 객체와 배열
  - 지역 변수
- 메소드의 이름
  - 메소드 영역에 로드 된 클래스
  - 파라미터
- 리턴 타입
  - 리턴 값
- 동적 할당
  - 연산
- 객체 참조가 저장
- static 변수
  - 위의 임시 값들이 저장
- final class 변수
  - 이것을 이용하여 Thread를 돌아가면서 쓸 수 있음

ARM에서  
동작하는 것  
PC는 IP레지스터를 씀

- PC register
- Native Method Stack
- Thread가 생성될 때마다 생성되는 영역
- 자바 외 언어로 작성된 코드를 위한 메모리 영역 (C, C++)

# CS/OS

## Programm vs Process

### Programm

- 파일이 저장장치에 저장되어 있지만, 메모리에는 올라가 있지 않은 정적인 상태
- 쉽게 말해, JUST CODE 덩어리

### Process

- 프로그램을 실행하고 있는 동적인 상태, 컴퓨터 메모리에 올라가있는 상태
- 쉽게 말해, 실행되고 있는 컴퓨터 프로그램
- 운영체제로 부터 자원 (CPU) 을 독자적으로 할당 받는작업의 단위



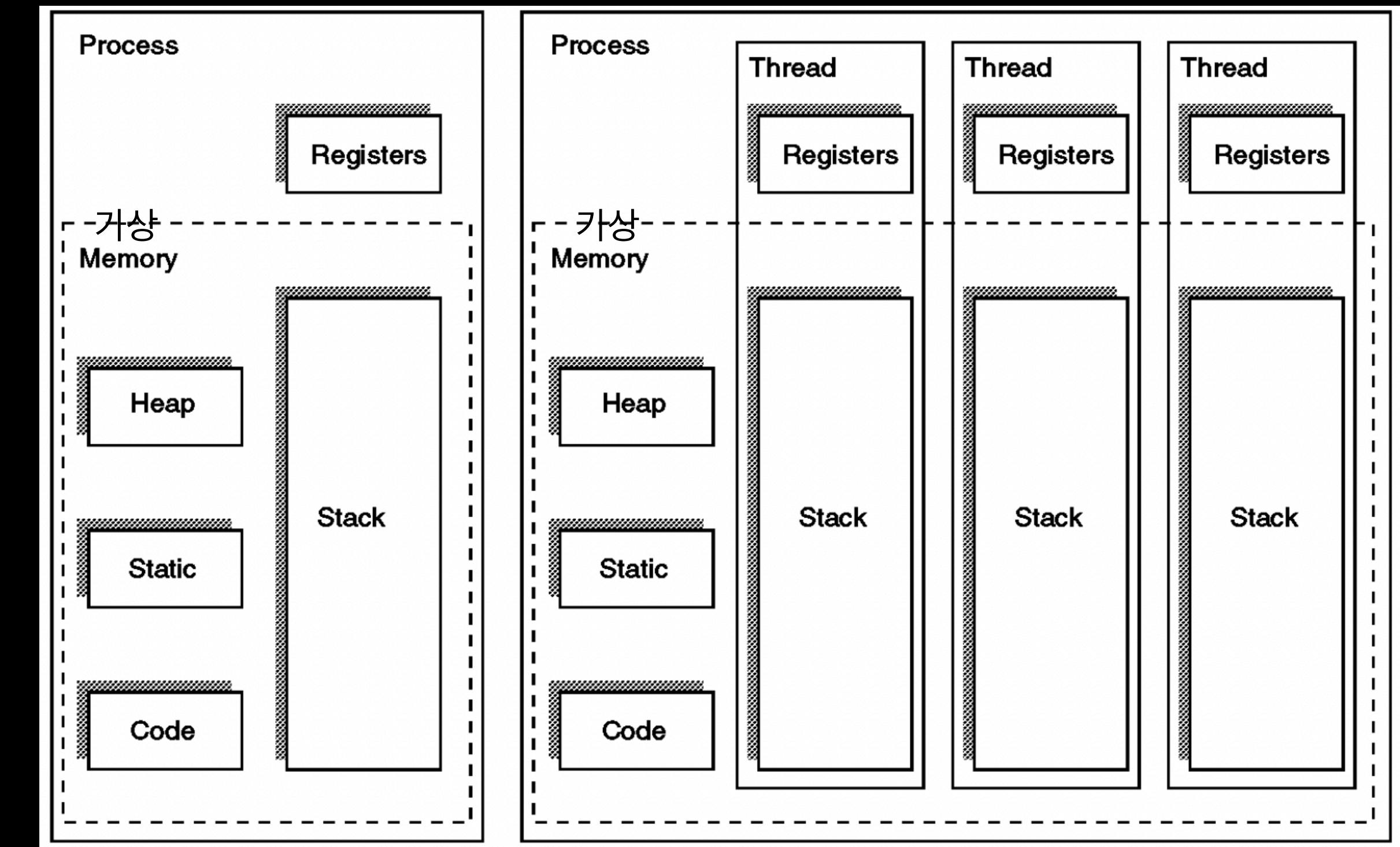
# CS/OS

## Process vs Thread

### Process(중대장)

- 여러 프로그램을 실행하고 있는 동적인 상태, 컴퓨터 메모리에 올라가있는 상태 (CPU의 주상화)
- 쉽게 말해, 실행되고 있는 컴퓨터 프로그램
- 운영체제로 부터 자원 (CPU) 을 독자적으로 할당 받는작업의 단위
- 프로세스는 4가지 영역의 가상 메모리 형식을 가지고 있음
- 프로세스는 다른 프로세스의 변수나 자료에 접근 X(메모리 공유 X)

윈도우 운영체제



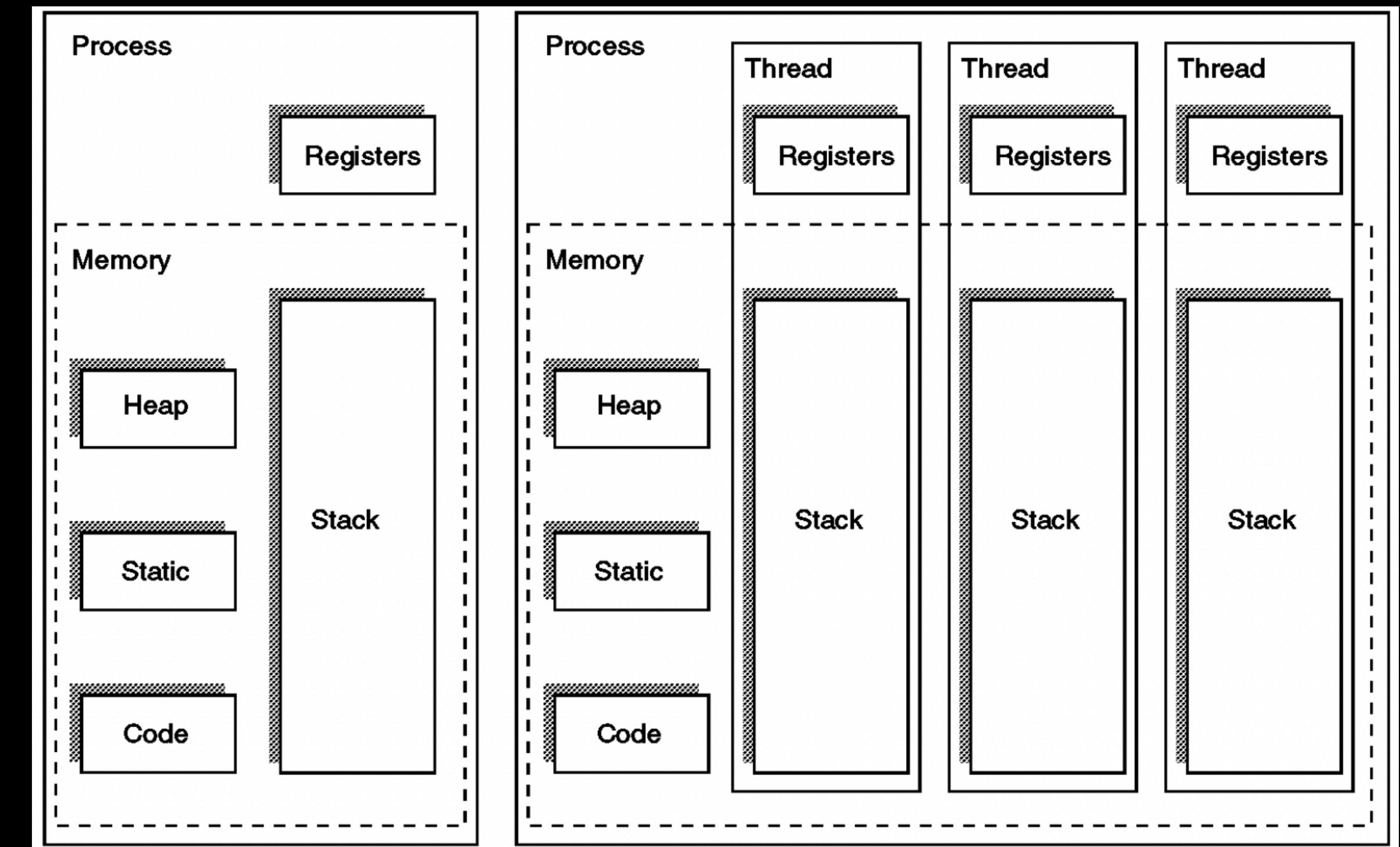
리눅스와 유닉스 에서는 프로세스와 쓰레드가 나누어 져있음

# CS/OS

## Process vs Thread

### Thread(중대원)

- 프로세스의 실행 단위
- 프로세스가 할당 받은 자원을 이용하는 실행의 단위
- 하나의 프로세스는 여러개의 Thread로 구성이 가능(윈도우 국한)
- Thread는 서로 공유가 가능
  - Stack 형식으로 할당된 메모리 영역을 따로 할당 받고
  - 나머지 3개의 형식으로 할당된 메모리를 공유함
  - 메모리를 공유 -> 오류 발생하면 다른 Thread도 강제 종료됨



# CS/OS

## MultiTasking vs MultiThread

- MultiTasking : 하나의 운영체제 안에서 여러 프로세스가 실행되는 것을 의미 한다  
(리눅스와 유닉스에서는 스레드 또한 독립적인 태스크라고 봄 그래서 멀티 태스킹이 쓰레드에도 포함 )
- MultiThread: 어떤 특정 작업을 여러 스레드를 사용하여 처리하는 것
- 하지만 동시에 실행되는것은 절대 아님 다만 속도가 엄청 빠르기때문에 동시에 처리하는것으로 느껴짐
- MultiThread의 장점 : 서로 공유하기때문에 메모리 자원을 아낌, 응답 시간이 엄청 빠름
- MultiThread의 단점: 스레드 하나가 프로세스 내 자원을 망쳐버린다면 모든 프로세스가 종료

동기화 문제 발생-프로그래머가 잘처리해야함 -> 진입장벽이 높다



# CS/OS

## Synchronisation Issue

- (CPU의 코어가 여러개 있을때 )병렬 처리 시 발생하는 이슈
- 여러개의 Thread는 CPU의 자원을 서로 받기위해 경쟁
- 각각의 Thread 중 어떤 것이 어떤 순서로 실행될 지 그 순서를 알 수 없음
- 이를 동기화 문제 (Synchronisation Issue) 라고함
- 즉 데이터 무결성이 깨지는 경우가 발생함.

```
// c = 4      : c = 4      ==> c = 4
// a = 5 + c  : a = 9      ==> a = 8
// c = 3      : c = 3      ==> c = 3
// b = a + c  : b = 12     ==> b = 11
```

-그림과 같은 경우 운영체제 내에서 기본적으로 지역 변수들이 같으면 다른 변수로 바꾸어 실행이 된다

-하지만 이러한 에러가 발생 할 확률이 많다 그렇기 때문에 프로그래머는 이러한 문제를 미리 방지하도록 프로그래밍을 해야함

# CS/OS

## Scheduling

- 운영체제는 시 분할로 메모리를 할당한다
- 운영체제에서 효율적으로 프로세스들을 처리하기 위해 하는것을 말함
- 각각의 준비된 Thread들을 Queue에 넣어 순서를 정하고, 순서에 따라 진행
- 순서는 Register를 통해 랜덤으로 배정 되는게 원칙
- 하지만 우선순위를 지정 해 줄 수도 sleep()을 통해 Queue에서 어떤 조건만큼 제외 가능

# Thread Method

## Runnable Interface

- Thread를 구현하기 위해서는 2가지 방법이 있다
- Thread 클래스를 상속 받는 법 / Runnable Interface 구현하는 방법
- 기본적으로 Runnable을 사용 (Thread 클래스를 상속 받으면 다른 클래스를 상속 받을 수 X)
- Thread 구현을 위해선 run()을 사용함
- run() vs start() 은 멀티 태스킹을 위한 중요한 메소드이다

```
class ThreadEx1 extends Thread{  
    public void run(){  
        //작업내용  
    }  
}  
  
ThreadEx1 ex1 = new ThreadEx1();  
ex1.start();
```

```
class RunnableImplements implements Runnable {  
    public void run() {  
        // 작업내용  
    } // Runnable인터페이스의 추상메서드 run()을 구현  
}
```

```
Runnable r = new RunnableImplements();  
Thread t = new Thread(r);  
t.start();
```

# Thread Method

## run( ) vs start( )

- run( ) vs start( ) 은 멀티 태스킹을 위한 중요한 메소드 이다
- run( )은 단순히 run( )클래스에 오버라이딩이 된 메소드를 호출해서 사용하는거
- start( )는 Thread 가 작업하는데 공간(호출스택)을 생성 후 run( )을 호출해서 그 공간 안에 저장
- 즉 start( ) 는 run( ) 의 메소드를 Runnable할수 있게 만들어주고 queue에 메소드들을 넣어줌
- 중대장 명령하달start())으로 소대원(run( ))들을 행정반이라는 공간(호출 스택)에 불러서 줄을 세움

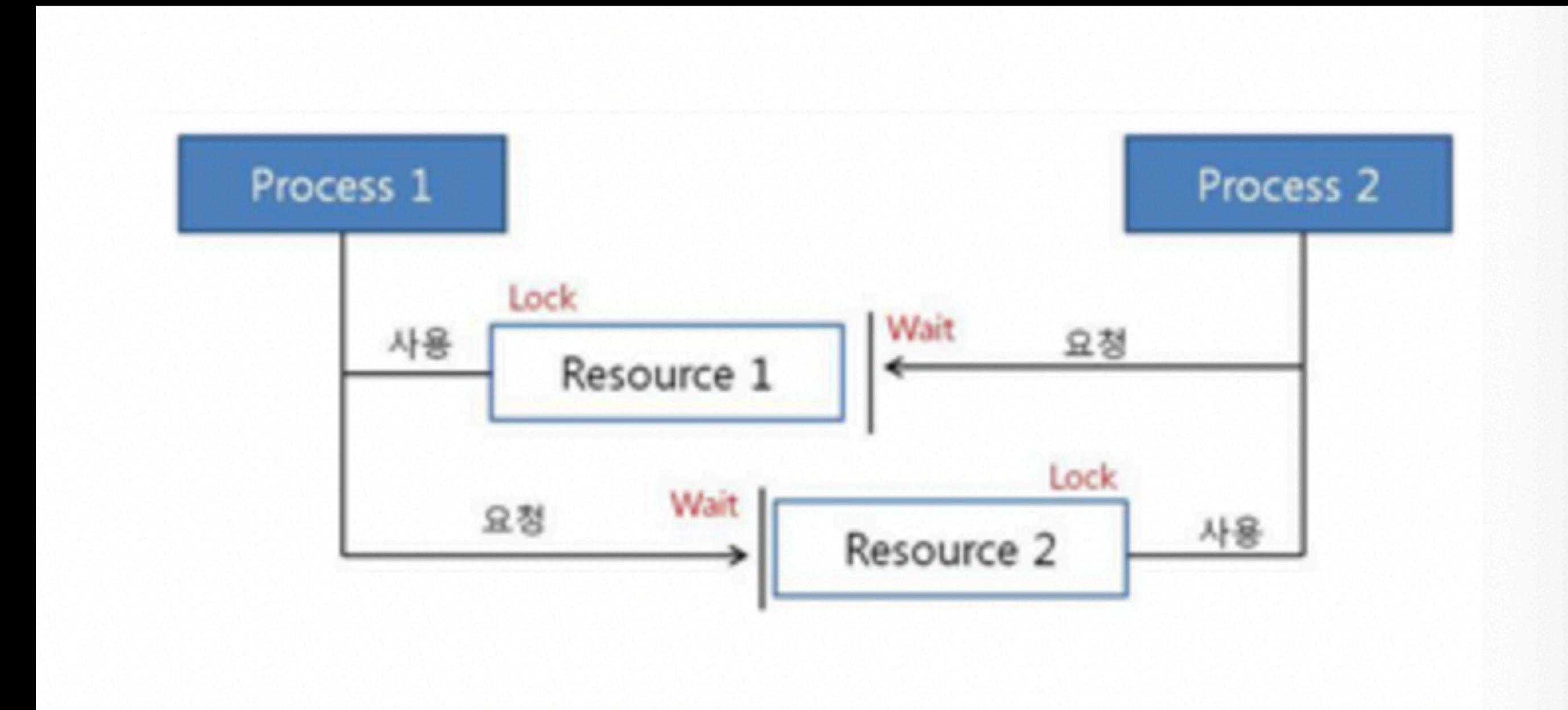
```
class ThreadEx1 extends Thread{  
    public void run(){  
        //작업내용  
    }  
}  
  
ThreadEx1 ex1 = new ThreadEx1();  
ex1.start();
```

```
class RunnableImplements implements Runnable {  
    public void run() {  
        // 작업내용  
    } // Runnable인터페이스의 추상메서드 run()을 구현  
}
```

```
Runnable r = new RunnableImplements();  
Thread t = new Thread(r);  
t.start();
```

# Dead Lock

- Process 또는 Thread가 자원을 얻지 못해 다음 처리를 하지 못하는 ‘교착 상태’
- 시스템적으로 한정된 자원(CPU)을 여러곳에서 사용하려 할 때 발생
- Process 1 과 2가 Resource 1,2 둘다 얻어야 할때 P1이 R1을 얻고 P2가 R2를 가졌다면 P1은 R2를 기달리고 P2는 R1을 무한정 기다리게 된다
- 이 상황을 Dead Lock이라고함



# Dead Lock

## 발생 조건

\*밑의 4가지 경우 중 하나라도 성립 안되게 만들면 DeadLock 해결

- 상호배제
  - 자원은 한번에 한프로세스/스레드만 사용할 수 있어야함
- 점유대기
  - 한번에 한개를 실행할때 다음 진행을 위해 대기하는 프로세스/스레드가 있어야함
- 비 선점
  - 한개의 프로세스 /스레드가 끝날 때까지 강제로 뺏을 수 없어야함
- 순환대기
  - 첫번째 프로세스/스레드는 다음으로 진행되는 프로세스/스레드가 점유한 자원을 대기하고 그다음은 그다음을 대기 .....마지막 자원은 맨 처음 자원을 대기해야함

# Dead Lock

## Dead Lock 처리

교착 상태 예방 및 회피	교착 상태가 되지 않도록 보장하기 위하여 교착 상태를 예방하거나 회피하는 프로토콜을 이용하는 방법
교착 상태 탐지 및 회복	교착 상태가 되도록 허용한 다음에 회복시키는 방법
교착 상태 무시	대부분의 시스템은 교착 상태가 잘 발생하지 않으며, 교착 상태 예방, 회피, 탐지, 복구하는 것은 비용이 많이 듈다.