

# Thread

# Thread

## ❖ Thread

- ✓ Process: 운영체제로부터 자원을 할당받아서 동작하는 독립된 프로그램으로 하나의 프로세스가 다른 프로세스에 영향을 미치지 않음
- ✓ Thread: 프로세스 내의 명령어 블록으로 시작점과 종료점을 가지는 일련된 하나의 작업 흐름으로 혼자서 동작할 수 없고 항상 Process 내에 속해서 동작해야 하며 순차적으로 동작하는 문장들의 단일 집합으로 다른 Thread 와 자원을 공유할 수 있으며 실행 중에 멈출 수 있으며 (제어권을 다른 Thread에게 넘김) 다른 Thread를 수행하는 것이 가능
- ✓ Task: 일 또는 작업이라는 말로 번역하며 프로세스와 Thread를 포함한 개념
- ✓ 프로세스는 제어권을 가지면 종료 될 때까지 제어권의 이동이 불가능하고 다른 프로세스와 자원을 공유하지 않으며 통신을 이용해서만 다른 프로세스와 자원을 공유할 수 있음
- ✓ java 에서는 java.lang 패키지에서 Thread를 위한 Runnable 인터페이스와 Thread, ThreadGroup, ThreadLocal, InheritableThreadLocal 클래스를 제공
- ✓ 1.5 이상의 버전에서는 Thread의 동기화를 위해서 java.util.concurrent 패키지 제공

# Thread

## ❖ Thread

### ✓ 메인(main) Thread

- 모든 자바 프로그램은 메인 Thread가 main() 메서드를 실행해서 시작
- main() 메서드의 첫 코드부터 아래로 순차적으로 실행
- 실행 종료 조건
  - ◆ 마지막 코드 실행
  - ◆ return 문을 만나면
- main Thread는 작업 Thread들을 만들어 병렬로 코드들 실행

# Thread

## ❖ Thread

### ✓ Thread 프로그래밍의 장점

- CPU의 사용률을 향상시킴
- 자원을 보다 효율적으로 사용
- 사용자에게 대한 응답성이 향상
- 작업이 분리되어 코드가 간결

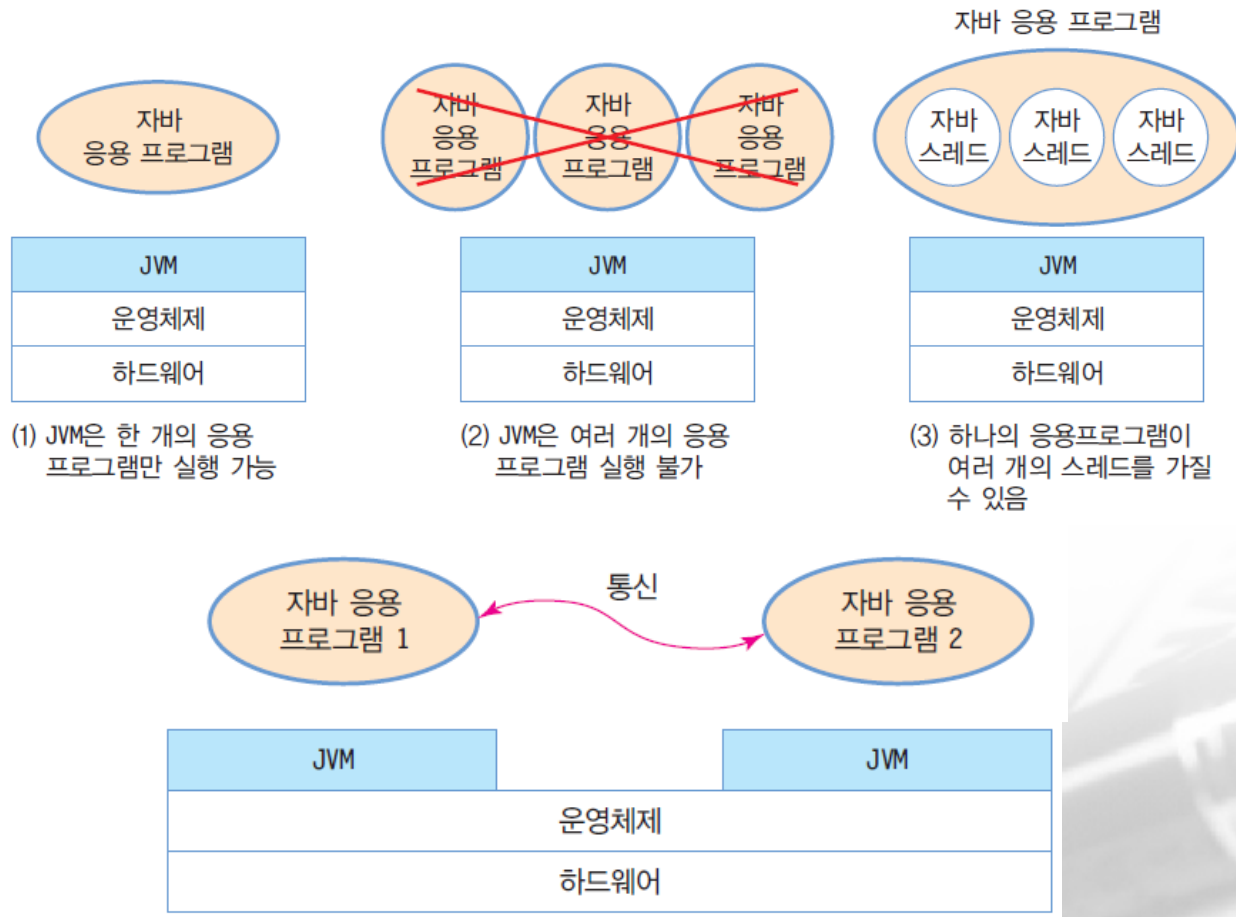
### ✓ Thread 프로그래밍의 단점

- 동기화나 교착상태 등을 고려해야 하기 때문에 프로그램이 어려워짐
- 너무 많은 Thread를 생성하면 스레싱 현상이 발생해서 성능이 저하 될 가능성이 있음



# Thread

## ❖ Thread



- ❖ 두 개의 자바 응용프로그램을 동시에 실행시키면 두 개의 JVM을 이용하고 응용프로그램은 서로 소켓 등을 이용하여 통신

# Thread

## ❖ Thread 클래스

- ✓ JDK는 작업 Thread를 생성해서 멀티 태스킹을 할 수 있도록 `java.lang.Runnable` 인터페이스와 `java.lang.Thread` 클래스를 제공
- ✓ Thread의 생성자
  - `Thread()`
  - `Thread(String s)`: Thread 이름을 설정해서 생성
  - `Thread(Runnable r)`: Runnable 을 implements 한 클래스 객체를 이용해서 생성
  - `Thread(Runnable r, String s)`:

# Thread

## ❖ Thread 클래스

### ✓ 메서드

- **static void sleep(long msec) throws InterruptedException:** msec에 지정된 밀리초 (milliseconds) 동안 현재 Thread를 대기시킴
- **String getName():** Thread의 이름을 반환
- **void setName(String s):** Thread의 이름을 s로 설정
- **int getPriority():** Thread의 우선 순위를 반환
- **void setPriority(int p):** Thread의 우선 순위를 p값으로 설정
- **boolean isAlive():** Thread가 시작되었고 아직 끝나지 않았으면 true를 그렇지 않으면 false를 반환
- **void join() throws InterruptedException :** 호출하는 Thread가 끝날 때까지 대기
- **void run():** Thread가 실행할 부분을 기술하는 메서드로 하위 클래스에서 오버라이딩 되어야 함
- **void start():** run 메서드의 내용을 Thread로 실행시켜 주는 메서드

# Thread

## ❖ Thread 클래스

### ✓ Thread를 생성하는 방법

- Thread 클래스를 상속받아 Thread를 생성
- Runnable 인터페이스를 implements 해서 생성

### ✓ Thread 클래스 이용

- Thread 클래스로부터 상속

```
class ThreadA extends Thread
{
    .....
    public void run()
    {
        .... // 상위 클래스인 Thread 클래스의 run() 메서드를 오버라이딩하여 Thread가
        수행하여야 하는 문장들을 기술
    }
    .....
}
```

- Thread 클래스에서 제공되는 run() 메서드를 오버라이딩하여 Thread의 동작을 기술
- Thread 생성 및 시작

```
ThreadA TA = new ThreadA();
TA.start();
```



# Thread

## ❖ Thread 클래스

- ✓ Thread 클래스를 상속받아 Thread를 생성

```
//0.5초 간격으로 Thread 자신의 이름을 10번 출력하는 Thread
public class ThreadTest extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                Thread.sleep(500);
                System.out.println(getName());
            }
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

Problems  
<terminated>  
스레드:0  
스레드:0  
스레드:1  
스레드:1  
스레드:2  
스레드:2  
스레드:2  
스레드:3  
스레드:3  
스레드:4  
스레드:4  
스레드:5  
스레드:5  
스레드:6  
스레드:6  
스레드:7  
스레드:7  
스레드:8  
스레드:8  
스레드:9  
스레드:9

Problems  
<terminated>  
스레드:0  
스레드:1  
스레드:2  
스레드:3  
스레드:4  
스레드:5  
스레드:6  
스레드:7  
스레드:8  
스레드:9  
스레드:0  
스레드:1  
스레드:2  
스레드:3  
스레드:4  
스레드:5  
스레드:6  
스레드:7  
스레드:8  
스레드:9

# Thread

## ❖ Thread 클래스

- ✓ Thread 클래스를 상속받아 Thread를 생성

```
public class Thread1 {  
    public static void main(String[] args) {  
        // 2개의 Thread를 생성해서 실행  
        ThreadTest t1 = new ThreadTest();  
        ThreadTest t2 = new ThreadTest();  
        t1.start();  
        t2.start();  
        // t1.run();  
        // t2.run();  
    }  
}
```

# Thread

## ❖ Thread

- ✓ Runnable 인터페이스를 이용하여 Thread를 생성할 수 있음
- ✓ Runnable 인터페이스에는 run() 메서드만 선언되어 있음

```
public interface Runnable {  
    public void run();  
}
```

- ✓ Runnable 인터페이스 이용

```
class RunnableB extends Applet implements Runnable  
{  
    public void run()  
    {  
        ..... // Runnable 인터페이스에 정의된 run() 메서드를  
        ..... // 오버라이딩하여 Thread가 수행할 문장들을 기술  
    }  
}  
  
RunnableB rb = new RunnableB(); // 객체 rb 생성  
Thread tb = new Thread(rb);  
// rb를 매개변수로 하여 Thread 객체 tb를 생성  
tb.start(); // Thread 시작
```

# Thread

## ❖ Thread

### ✓ Runnable 인터페이스를 이용

//0.2초 간격으로 0부터 9까지 출력하는 Thread

class RunnableTest implements Runnable

```
{
    public void run()
    {
        try
        {
            for (int i=0 ; i<10 ; i++)
            {
                Thread.sleep(200);
                System.out.println("Thread :" + i);
            }
        }
        catch (InterruptedException e )
        {
            e.printStackTrace();
        }
    }
}
```

| Problems     | Problems     |
|--------------|--------------|
| <terminated> | <terminated> |
| 스레드 : 0      | 스레드 : 0      |
| 스레드 : 0      | 스레드 : 1      |
| 스레드 : 1      | 스레드 : 2      |
| 스레드 : 1      | 스레드 : 3      |
| 스레드 : 2      | 스레드 : 4      |
| 스레드 : 2      | 스레드 : 5      |
| 스레드 : 3      | 스레드 : 6      |
| 스레드 : 3      | 스레드 : 7      |
| 스레드 : 4      | 스레드 : 8      |
| 스레드 : 4      | 스레드 : 9      |
| 스레드 : 5      | 스레드 : 0      |
| 스레드 : 5      | 스레드 : 1      |
| 스레드 : 6      | 스레드 : 2      |
| 스레드 : 6      | 스레드 : 3      |
| 스레드 : 7      | 스레드 : 4      |
| 스레드 : 7      | 스레드 : 5      |
| 스레드 : 8      | 스레드 : 6      |
| 스레드 : 8      | 스레드 : 7      |
| 스레드 : 9      | 스레드 : 8      |
| 스레드 : 9      | 스레드 : 9      |

# Thread

## ❖ Thread

- ✓ Runnable 인터페이스를 이용

```
public class Thread2
{
    public static void main(String args[])
    {
        RunnableTest Obj = new RunnableTest();
        // Runnable 인터페이스 객체로 r을 생성
        Thread th = new Thread(Obj);
        // r을 매개 변수로 하여 Thread 객체 th를 생성
        th.start();
    }
}
```

# Thread

## ❖ Thread 수명 주기

### ✓ Thread 상태

- NEW: Thread가 생성되었지만 Thread가 아직 실행할 준비가 되지 않았음
  - RUNNABLE: Thread가 JVM에 의해 실행되고 있거나 실행 준비되어 스케줄링을 기다리는 상태
  - WAITING: 다른 Thread가 notify(), notifyAll()을 불러주기를 기다리고 있는 상태.
  - TIMED\_WAITING: Thread가 sleep(n) 호출로 인해 n 밀리초 동안 잠을 자고 있는 상태
  - BLOCK: Thread가 I/O 작업을 요청하면 JVM이 자동으로 이 Thread를 BLOCK 상태로 만듦
  - TERMINATED: Thread가 종료된 상태
- ✓ Thread 상태는 JVM에 의해 기록되고 관리됨
  - ✓ run() 메서드가 종료되면 Thread는 종료
  - ✓ 한번 종료한 Thread는 다시 시작시킬 수 없음
  - ✓ Thread에서 다른 Thread를 강제 종료시킬 수 있음

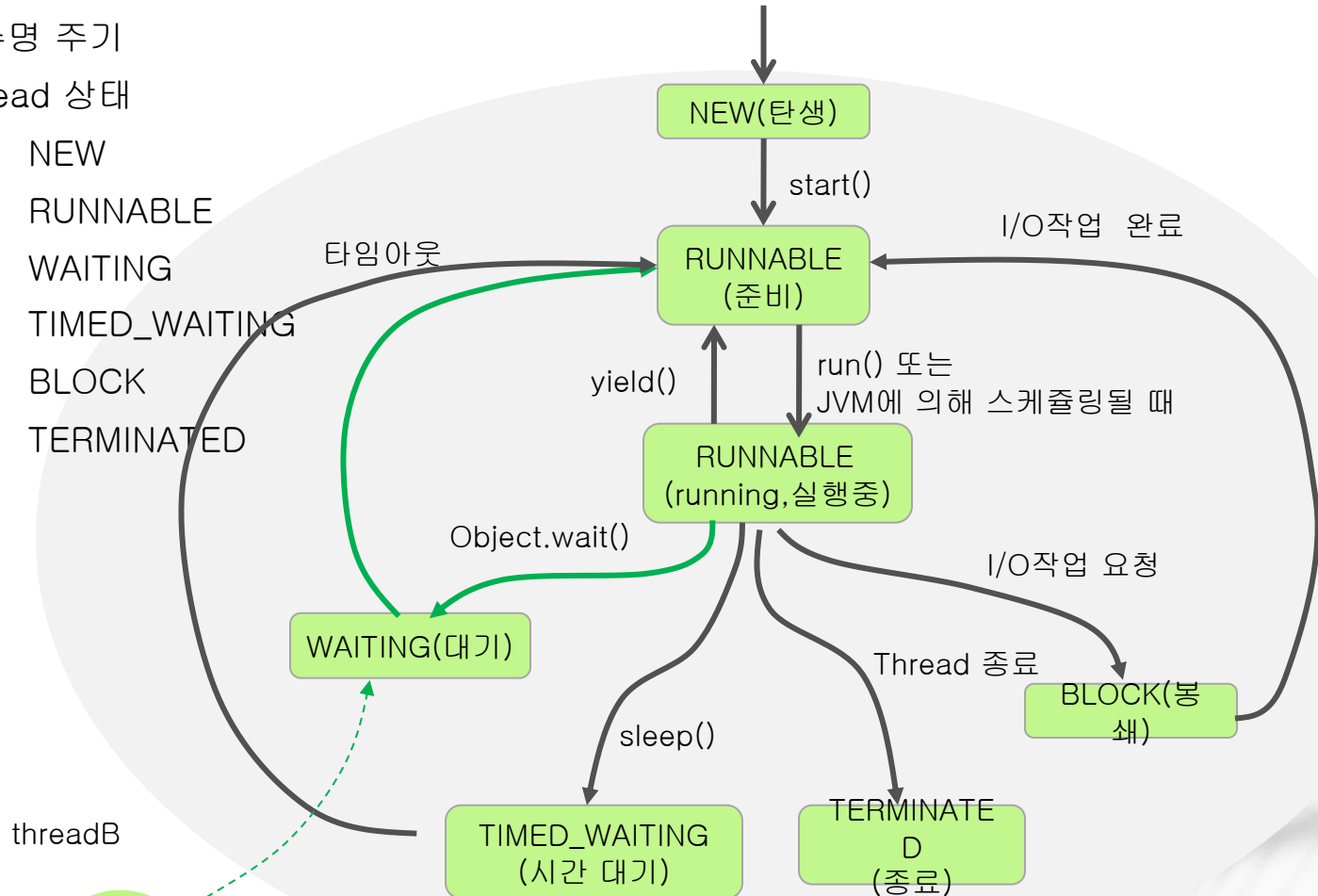
# Thread

threadA = new Thread()

## ❖ Thread 수명 주기

### ✓ Thread 상태

- NEW
- RUNNABLE
- WAITING
- TIMED\_WAITING
- BLOCK
- TERMINATED



\*\* wait(), notify(), notifyAll()은 Thread의 메서드가 아니며 Object의 메서드

# Thread

## ❖ Daemon Thread

- ✓ 백그라운드 상태에서 대기하고 있다가 처리할 요청이 발생하거나 조건 상황이 맞으면 작업을 수행하는 Thread
- ✓ 동일한 프로세스 안에서 다른 Thread의 수행을 돕는 보조적인 Thread로 다른 Thread를 서비스 해주면서 Daemon Thread가 아닌 Thread가 모두 종료되면 자신도 자동으로 종료되는 Thread
- ✓ 가비지 컬렉터나 메인 Thread가 대표적인 데몬 Thread
- ✓ Thread 객체를 생성하고 `setDaemon(true)`으로 설정하면 됨
- ✓ Thread가 시작하기 전에 설정해야 함
- ✓ 일반적으로 데몬 Thread는 특정 조건을 만족하면 작업을 수행하고 다시 대기하도록 작성
- ✓ 응용 프로그램에서 자동 저장 기능이나 온라인 게임에서 자동으로 데이터를 전송하는 작업을 데몬 Thread로 작성
- ✓ 데몬 Thread안에서 Thread를 생성한다면 이 Thread 역시 데몬 Thread



# Thread

## ❖ Daemon Thread

### ✓ 설정

```
public class Thread3{
    public static void main(String args[]){
        Thread t = new Thread(){
            public void run()
            {
                try{
                    System.out.println("데몬 Thread 시작");
                    sleep(10000);
                    System.out.println("데몬 Thread 종료");
                }
                catch (Exception e)
                { }
            }
        };
        //t.setDaemon(true);
        t.start();
        System.out.println("메인 함수 종료");
    }
}
```

# Thread

## ❖ Daemon Thread

### ✓ AutoSave

```
public class AutoSave implements Runnable {
    static boolean autoSave = false;

    public void run() {
        while (true) {
            try {
                Thread.sleep(3 * 1000);
            } catch (InterruptedException e) {
            }

            if (autoSave) {
                autoSave();
            }
        }
    }

    public void autoSave() {
        System.out.println("작업파일이 자동저장되었습니다.");
    }
}
```

# Thread

## ❖ Daemon Thread

### ✓ AutoSave

```
public class AutoSaveMain {  
    public static void main(String[] args) {  
        Thread t = new Thread(new AutoSave());  
        t.setDaemon(true);           // 이 부분이 없으면 종료되지 않는다.  
        t.start();  
  
        for(int i=1; i <= 20; i++) {  
            try{  
                Thread.sleep(1000);  
            } catch(InterruptedException e) {}  
            System.out.println(i);  
  
            if(i==5)  
                AutoSave.autoSave = true;  
        }  
        System.out.println("프로그램을 종료합니다.");  
    }  
}
```

# Thread

## ❖ Thread 우선순위

- ✓ 2개 이상의 Thread가 동작 중일 때 우선 순위를 부여하여 우선 순위가 높은 Thread에게 실행의 우선권을 부여 가능
- ✓ 우선 순위를 지정하기 위한 상수를 제공
  - `static final int MAX_PRIORITY`                      가장 높은 우선 순위
  - `static final int MIN_PRIORITY`                      가장 낮은 우선 순위
  - `static final int NORM_PRIORITY`                      보통의 우선 순위
- ✓ Thread 우선 순위 접근자 메서드
  - `void setPriority(int priority)`
  - `int getPriority()`
- ✓ `main()` Thread의 우선 순위 값은 초기에 `NORM_PRIORITY`
- ✓ 자식 Thread는 부모 Thread와 동일한 우선순위 값을 가지고 탄생
- ✓ JVM의 스케줄링 정책
  - 철저한 우선 순위 기반
  - 가장 높은 우선 순위의 Thread가 우선적으로 스케줄링
  - 동일한 우선 순위의 Thread는 돌아가면서 스케줄링(라운드 로빈)
- ✓ 파일 입출력이나 네트워크 전송 등의 CPU 사용이 작은 작업은 우선순위를 낮게 하고 연산작업 처럼 CPU 사용이 빈번한 경우에는 우선순위를 높여주는 것을 권장

# Thread

## ❖ Thread 우선순위

```
public class ThreadPriority extends Thread {
    ThreadPriority(String str)
    {
        super(str);
    }
    public void run()
    {
        try {
            for (int i=0 ; i<10 ; i++) {
                Thread.sleep(1000);
                System.out.println(getName() + i + "번째 수행");
            }
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

# Thread

## ❖ Thread 우선순위

```
public class PriorityMain
{
    public static void main(String args[])
    {
        ThreadPriority t1 = new ThreadPriority("우선 순위가 낮은 Thread");
        ThreadPriority t2 = new ThreadPriority("우선 순위가 높은 Thread");
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
    }
}
```

# Thread

## ❖ Thread Group

- ✓ 서로 관련된 Thread를 하나의 그룹으로 묶어서 사용하기 위한 개념
- ✓ Thread 클래스의 배열이나 컬렉션을 이용하는 것을 권장 - 몇몇 메서드가 제대로 동작하지 않음
- ✓ ThreadGroup 객체를 생성한 후 Thread 클래스의 생성자에 대입
- ✓ JVM은 기본적으로 main Thread 그룹과 system Thread 그룹을 생성

# Thread

## ❖ Thread Group

```
public class GroupThread {  
    public static void main(String[] args) {  
        ThreadGroup main = Thread.currentThread().getThreadGroup();  
        ThreadGroup group1 = new ThreadGroup("그룹1");  
        ThreadGroup group2 = new ThreadGroup("그룹2");  
  
        Thread th1 = new Thread(group1, "Thread1");  
        Thread th2 = new Thread("Thread2");  
        Thread th3 = new Thread(group2, "Thread3");  
  
        th1.start();  
        th2.start();  
        th3.start();  
  
        System.out.println("실행 중인 Thread 그룹의 개수: " + main.activeGroupCount());  
        System.out.println("실행 중인 Thread 개수: " + main.activeCount());  
    }  
}
```



# Thread

## ❖ Thread 종료

### ✓ 스스로 종료

- run() 메서드 리턴

### ✓ 강제 종료

- Thread의 interrupt() 메서드를 호출하면 Thread에 InterruptedException이 발생됨
- Thread의 run 메서드 안에서 예외처리 구문을 만들고 return 을 하면 됨



# Thread

## ❖ Thread 종료

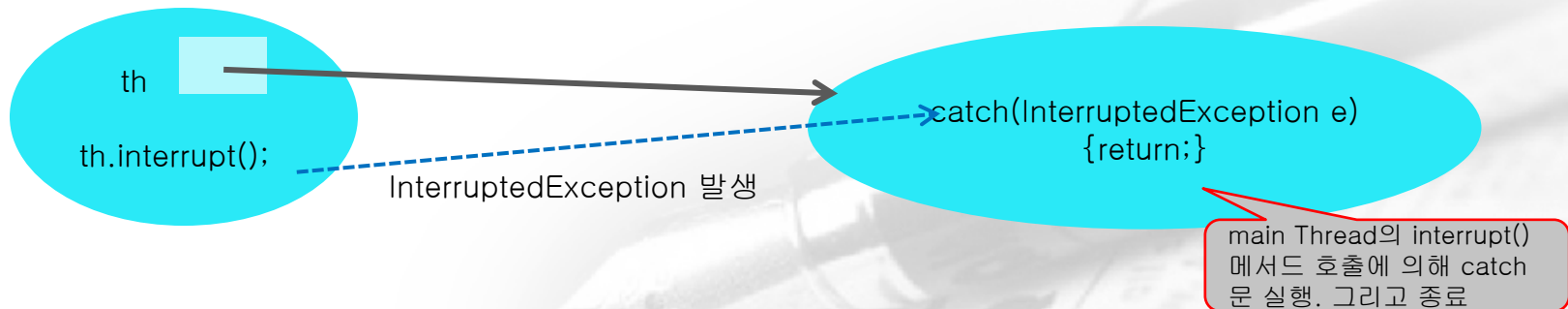
```
public static void main(String [] args) {  
    TimerThread th = new TimerThread();  
    th.start();  
  
    th.interrupt(); // TimerThread 강제 종료  
}
```

main() Thread

```
class TimerThread extends Thread {  
    int n = 0;  
    public void run() {  
        while(true) {  
            System.out.println(n); // 화면에 카운트 값 출력  
            n++;  
            try {  
                sleep(1000);  
            }  
            catch (InterruptedException e){  
                return; // 예외를 받고 스스로 리턴하여 종료  
            }  
        }  
    }  
}
```

만일 return 하지 않으면  
Thread는 종료하지 않음

TimerThread Thread



# Thread

## ❖ Thread 종료

```
public class ThreadInterrupt extends Thread {
    ThreadInterrupt(String str)
    {
        super(str);
    }
    public void run()
    {
        try {
            for (int i=0 ; i<10 ; i++) {
                Thread.sleep(1000);
                System.out.println(getName() + i + "번째 수행");
            }
        } catch (InterruptedException e)
        {
            System.out.println("Thread 강제 종료");
            return;
        }
    }
}
```

# Thread

## ❖ Thread 종료

```
public class InterruptMain {  
    public static void main(String args[]){  
        ThreadInterrupt th = new ThreadInterrupt("Thread");  
        th.start();  
        try{  
            Thread.sleep(3000);  
        }  
        catch(InterruptedException e)  
        {}  
        th.interrupt();  
    }  
}
```

# Thread

## ❖ 실행 제어 메서드

- ✓ void join(long millis, int nanos): 지정된 시간 동안 Thread를 실행
- ✓ void suspend(): Thread를 일시 정지시키는 메서드로 resume()에 의해 다시 시작
- ✓ void resume(): 일시 정지된 Thread를 다시 시작
- ✓ void yield(): 다른 Thread에게 실행 상태를 양보하고 자신은 준비 상태로 변경



# Thread

## ❖ 실행 제어 메서드 - yield

```
class ThreadA extends Thread {
    public boolean stop = false;
    public boolean work = true;
    public void run() {
        while(!stop) {
            if(work) {
                System.out.println("ThreadA 작업 내용");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } else {
                Thread.yield();
            }
        }
        System.out.println("ThreadA 종료");
    }
}
```

ThreadA 작업 내용  
ThreadB 작업 내용  
ThreadB 작업 내용  
ThreadA 작업 내용  
ThreadA 작업 내용  
ThreadB 작업 내용  
ThreadB 작업 내용  
ThreadB 작업 내용

# Thread

## ❖ 실행 제어 메서드 – yield

```
class ThreadB extends Thread {  
    public boolean stop = false;  
    public boolean work = true;  
    public void run() {  
        while(!stop) {  
            if(work) {  
                System.out.println("ThreadB 작업 내용");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            } else {  
                Thread.yield();  
            }  
        }  
        System.out.println("ThreadB 종료");  
    }  
}
```

# Thread

## ❖ 실행 제어 메서드 - yield

```
public class ThreadYield {  
    public static void main(String[] args) {  
        ThreadA threadA = new ThreadA();  
        ThreadB threadB = new ThreadB();  
        threadA.start();  
        threadB.start();  
  
        try { Thread.sleep(3000); } catch (InterruptedException e) {}  
        threadA.work = false;  
  
        try { Thread.sleep(3000); } catch (InterruptedException e) {}  
        threadA.work = true;  
  
        try { Thread.sleep(3000); } catch (InterruptedException e) {}  
        threadA.stop = true;  
        threadB.stop = true;  
    }  
}
```



# Thread

## ❖ 실행 제어 메서드 – join

1~100 합: 55

```
class SumThread extends Thread {  
    private long sum;  
  
    public long getSum() {  
        return sum;  
    }  
  
    public void setSum(long sum) {  
        this.sum = sum;  
    }  
  
    public void run() {  
        for(int i=1; i<=100; i++) {  
            sum+=i;  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

# Thread

## ❖ 실행 제어 메서드 – join

```
public class ThreadJoin {  
    public static void main(String[] args) {  
        SumThread sumThread = new SumThread();  
        sumThread.start();  
        try {  
            sumThread.join(1000);  
        } catch (InterruptedException e) {}  
        System.out.println("1~100 합: " + sumThread.getSum());  
    }  
}
```

# Thread

## ❖ Multi Thread

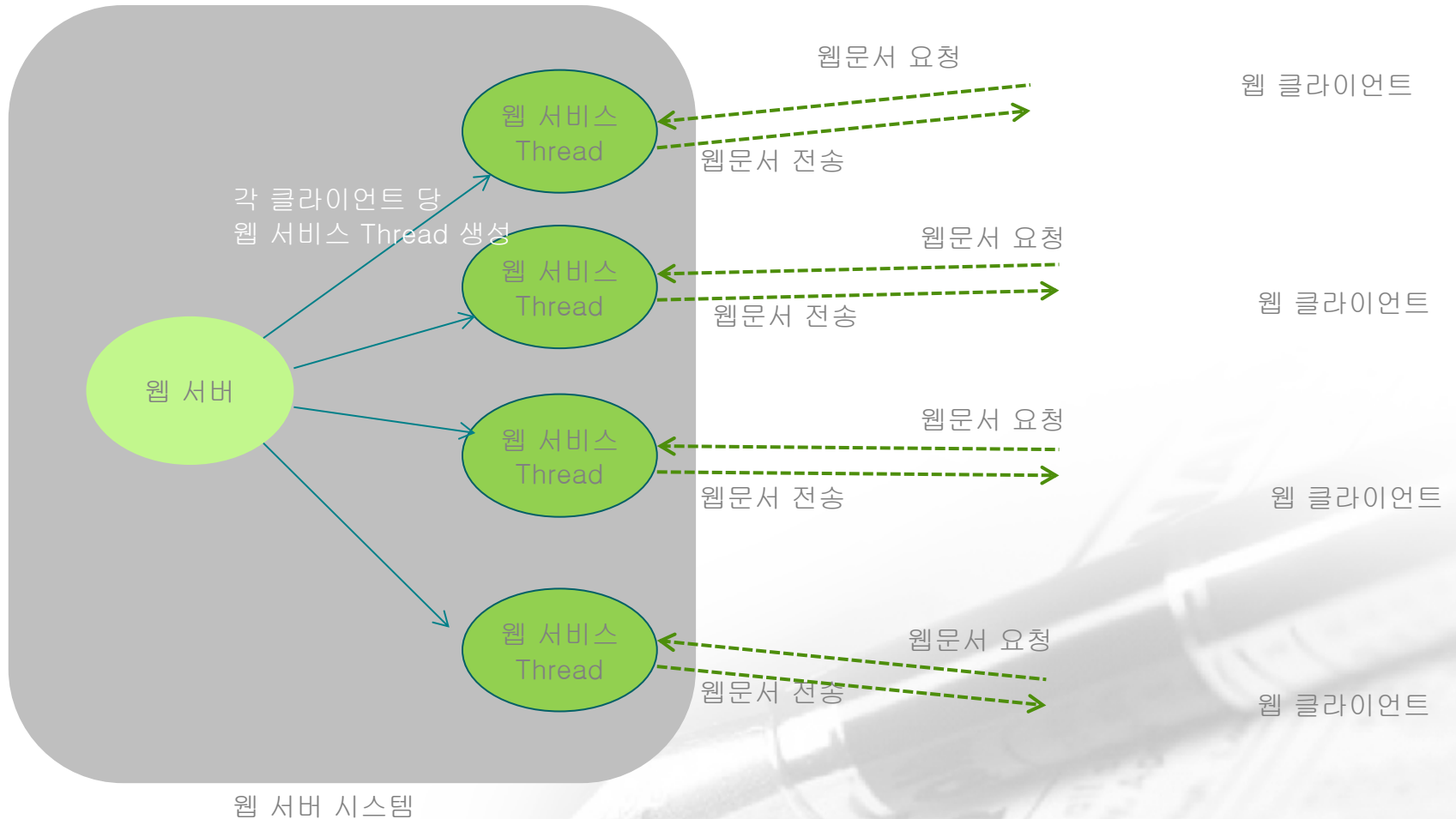
- ✓ 어느 한 시점에 2개 이상의 Thread가 동작 중인 경우
- ✓ 하나의 프로그램에서 여러 개의 Thread를 사용하게 되면 프로그램의 성능은 우수해 질 가능성이 높지만 하나의 Thread가 실행 중인 경우보다 항상 성능이 우수한 것은 아님
- ✓ 멀티 Thread 프로그램의 경우 Thread 사이의 전환이 일어날 때 현재 Thread의 정보를 저장하고 새로운 Thread의 정보를 읽어와야 하는데(Context Switching – 문맥교환) 이 작업으로 인한 오버헤드가 발생하기 때문에 성능이 저하될 수 도 있음
- ✓ CPU를 무조건 사용해야 하는 연산작업을 하나의 CPU를 사용하는 컴퓨터에서 멀티 Thread로 수행한다면 속도는 저하될 가능성이 높음
- ✓ 멀티 Thread 프로그램은 공유 자원 문제도 고려
- ✓ 장점
  - 하나의 장시간을 요하는 작업 때문에 뒤에 있는 간단한 작업이 기다릴 필요가 없음
  - 태스크의 대기 시간을 효율적으로 활용할 수 있음
- ✓ 단점
  - 메모리의 사용량이 증가
  - 처리량이 감소
  - 동시성 문제 발생

# Thread

- ❖ 멀티 Thread에서 발생할 수 있는 공유 자원 사용의 문제점
  - ✓ 임계 영역(critical section)에서 공유 자원의 상호 배제 문제
    - 공유 자원을 사용하는 코드 영역이 임계 영역
    - 임계 영역에서는 하나의 Thread가 사용 중인 공유 자원을 수정할 수 없도록 상호 배타 (Mutual Exclusion)적으로 실행될 수 있도록 작성되어야 함
    - 하나의 Thread가 사용 중인 공유 자원에 쓰기를 수행하면 이전 Thread가 갱신한 내용이 사라질 수 있음
  - ✓ Dead Lock 문제
    - 멀티 Thread를 사용할 때 주의할 점 중의 하나로 Thread를 잘못 만들면 프로그램의 수행이 이루어 지지 않고 무한대기하는 Dead Lock을 발생할 수 있음
    - 무한 대기 상태에 빠질 수 있음
  - ✓ 생산자와 소비자 문제
    - 공유 자원을 사용할 때 순서의 문제
  - ✓ 무한 루프 발생
    - Map에 여러 개의 Thread가 put을 하면 무한 루프 발생

# Thread

## ❖ Multi Thread

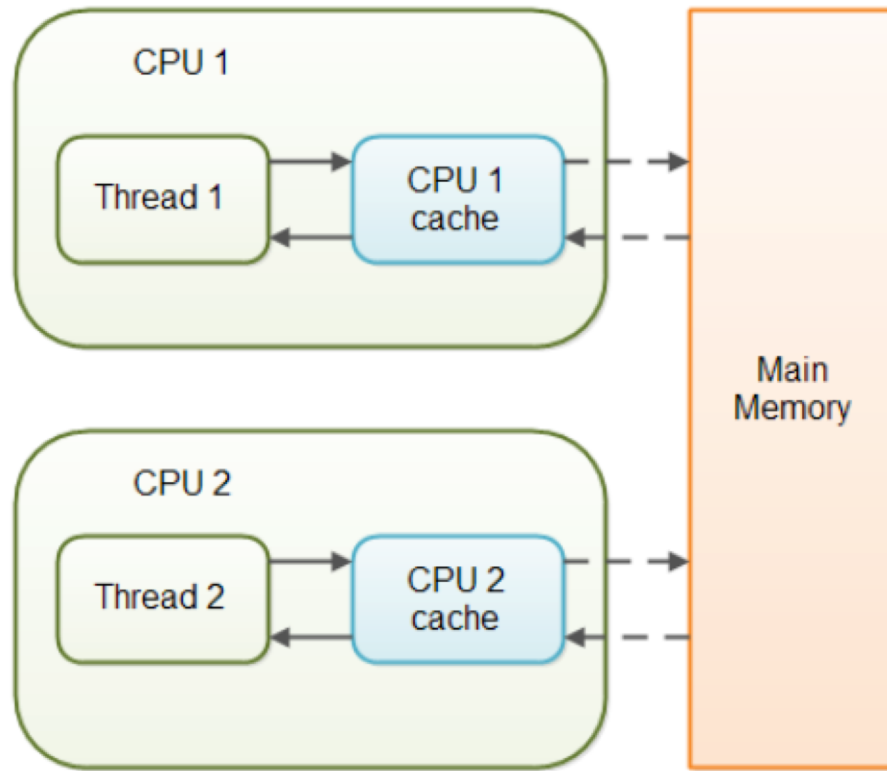


# Thread

- ❖ 공유 자원의 동시 수정 문제 해결 방안 – 상호배제(Mutual Exclusion)
  - ✓ 변수에 volatile을 적용
  - ✓ 공유 자원에 접근하는 메서드의 결과형 앞에 synchronized 추가 – 메서드를 동기화
  - ✓ 공유 자원을 사용하는 영역을 synchronized(객체명)의 블록으로 지정 – 블록 내부의 내용만 동기화

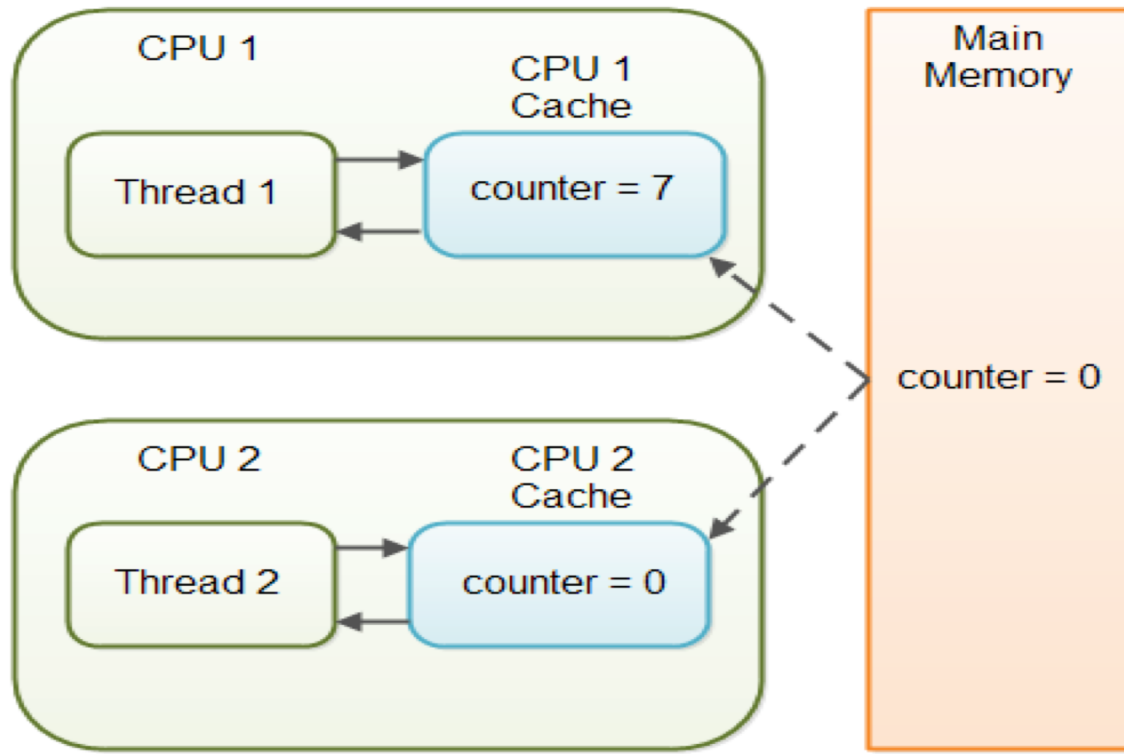
# Thread

- ❖ volatile: 특정 변수의 읽고 쓰기 작업에 대해 원자성을 보장해주는 예약어
  - ✓ CPU Cache를 이용하지 않고 Main Memory의 데이터를 직접 사용하기 위한 예약어
  - ✓ 스레드에서 메모리의 데이터를 사용하는 형태



# Thread

- ❖ volatile: 특정 변수의 읽고 쓰기 작업에 대해 원자성을 보장해주는 예약어
  - ✓ 여러 개의 스레드가 하나의 공유 자원을 수정하는 경우





# Thread

## ❖ 메서드 동기화

```
public class ShareData implements Runnable {
    private int result;
    private int idx;
    public int getResult() {
        return result;
    }
    private void sum() {
        for (int i = 0; i < 1000; i++) {
            idx++;
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
            }
            result += idx;
        }
    }
    @Override
    public void run() {
        sum();
    }
}
```

# Thread

## ❖ 메서드 동기화

```
public class ShareDataMain {  
    public static void main(String[] args) {  
        ShareData share = new ShareData();  
        Thread th1 = new Thread(share);  
        Thread th2 = new Thread(share);  
  
        th1.start();  
        th2.start();  
  
        try{  
            Thread.sleep(10000);  
        }  
        catch(Exception e){}  
  
        System.out.println("res:" + share.getResult());  
    }  
}
```

# Thread

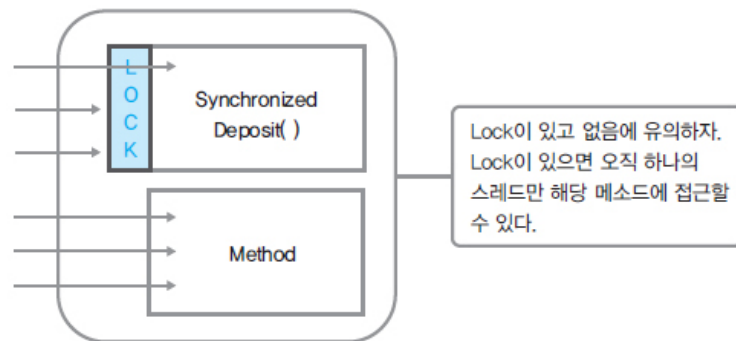
## ❖ 동기화

### ✓ synchronized method

- sum 메서드에 synchronized를 붙이지 않으면 수행 결과가 이상한 값을 출력하는 경우가 발생하는데 이러한 상황이 발생하는 이유는 2개의 Thread가 idx 값을 공유해서 수정하기 때문인데 하나의 Thread가 idx의 값을 수정하기 위해서 사용 중인데 다른 Thread가 idx를 수정해서 발생하는 현상
- 메서드의 수행이 종료될 때 까지 다른 Thread가 끼어들지 못하도록 할 수 있는데 이 때 메서드의 return type 앞에 synchronized를 붙이면 메서드가 호출 중에서는 다른 Thread가 이 메서드를 수행할 수 없게 됨

### ✓ synchronized block

- synchronized(공유 객체){ 내용; }을 작성하면 블록 내에서는 공유 객체에 lock을 설정해서 동시에 수정할 수 없도록 만들어 줌
- 하나의 Thread가 블록 내에 진입을 하게 되면 다른 Thread는 블록 안에 진입할 수 없게 됨



# Thread

## ❖ 동기화 - 동기화 블록

```
private void sum() {  
    for (int i = 0; i < 10000; i++) {  
        synchronized (this) {  
            idx++;  
            try {  
                Thread.sleep(1);  
            } catch (InterruptedException e) {  
            }  
            result += idx;  
        }  
    }  
}
```

# Thread

## ❖ 생산자와 소비자 문제

- ✓ 생산자와 소비자는 동시에 자원을 사용할 수 있지만 생산자가 자원을 생산해 주어야만 소비자가 자원을 소비할 수 있음
- ✓ 소비자가 생산자가 자원을 생산해주기 전에 자원을 사용하려고 하면 자원이 없어서 문제가 발생하는데 이러한 문제를 해결해주기 위해서 `java.lang.Object` 클래스에 Thread 사이의 통신을 위한 3개의 메서드를 제공
  - `wait()` 메서드: Thread 수행 중 이 메서드를 만나면 가지고 있는 lock을 양보하고 대기 상태로 진입
    - ◆ `void wait()` throws `InterruptedException` – 계속 대기 상태
    - ◆ `void wait(long msec)` throws `InterruptedException` – 대기 상태
    - ◆ `void wait(long msec, int nsec)` throws `InterruptedException` – 계속 대기 상태
      - msec와 nsec는 대기 시간
  - `notify()`: 대기 상태의 Thread중에서 하나의 Thread 를 수행
  - `notifyAll()`: 대기 상태의 모든 Thread 를 수행
    - ◆ 위의 메서드는 `synchronized` 블록이나 메서드 안에서만 사용이 가능한데 만일 위 메서드를 사용하는 영역이 `synchronized` 블록이나 메서드가 아니면 `java.lang.IllegalMonitorStateException`이 발생

# Thread

## ❖ 생산자와 소비자 문제 발생

```
import java.util.*;
class Product{
    List<Character> vec;
    Product(){
        vec=new ArrayList<Character>();
    }
    public void put(char ch){
        vec.add(ch);
        System.out.print("창고에 제품"+ch+"가 입고 되었습니다.\n");
        try{
            Thread.sleep(1000);
        }
        catch (Exception e){ }
        System.out.print("재고수량:"+vec.size()+"\n");
    }
    public char get(){
        Character ch=(Character)vec.remove(0);
        System.out.print("창고에서 제품"+ch+"가 출고 되었습니다"+"\n");
        System.out.print("재고수량:"+vec.size()+"\n");
        return ch.charValue();
    }
}
```

# Thread

## ❖ 생산자와 소비자 문제 발생

```
class Producer extends Thread
{
    private Product myW;
    public Producer(Product vec)
    {
        myW=vec;
    }
    public void run()
    {
        for(char ch='A'; ch<='Z'; ch++)
        {
            System.out.println("생산자가 제품"+ch+"을(를) 생산했습니다.");
            myW.put(ch);
        }
    }
}
```

# Thread

## ❖ 생산자와 소비자 문제 발생

```
class Customer extends Thread
{
    private Product myW;
    public Customer(Product vec)
    {
        myW = vec;
    }
    public void run()
    {
        char ch;
        for(int i=1; i<=26; i++)
        {
            ch=myW.get();
            System.out.println("소비자가 제품"+ch+"을(를) 소비했습니다.");
        }
    }
}
```



# Thread

## ❖ 생산자와 소비자 문제 발생

```
public class ProducerConsumerProblem
{
    public static void main(String args[])
    {
        Product Obj=new Product();
        Producer p=new Producer(Obj);
        Customer c=new Customer(Obj);

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

# Thread

## ❖ 생산자와 소비자 문제 해결

```
import java.util.*;
class Product{
    ArrayList <Character> vec;
    Product(){
        vec=new ArrayList<Character>();
    }
    synchronized void put(char ch){
        vec.add(ch);
        System.out.print("창고에 제품"+ch+"가 입고 되었습니다.\n");
        try{ Thread.sleep(1000); }
        catch (Exception e){}
        System.out.print("창고에 제품"+ch+"가 입고 되었습니다.\n");
        System.out.print("재고수량:"+vec.size()+"\n");
        notifyAll();
    }
}
```

# Thread

## ❖ 생산자와 소비자 문제 해결

```
synchronized char get(){
    try{
        if(vec.size()==0){
            wait();
        }
    }

    catch(InterruptedException E){}

    Character ch=(Character)vec.remove(0);
    try{
        Thread.sleep(1000);
    }
    catch (Exception e)
    {}
    System.out.print("창고에서 제품"+ch+"가 출고 되었습니다"+"Wn");
    System.out.print("재고수량:"+vec.size()+"Wn");
    return ch.charValue();
}
}
```

# Thread

## ❖ DeadLock

- ✓ 결코 발생할 수 없는 사건을 무한정 기다리는 것
- ✓ 대부분 동기화된 메서드에서 다른 동기화된 메서드를 호출하거나 동기화된 블록을 만들기 때문
- ✓ DeadLock을 예방하기 위해서는 동기화된 메서드 또는 블록에서 다른 동기화된 메서드 또는 블록을 호출하지 않는 것이 바람직

# Thread

## ❖ DeadLock

```
import java.util.List;
public class ResourceLocker implements Runnable {
    private String name;
    private List<String> fromList;
    private List<String> toList;

    public ResourceLocker(String name, List<String> fromList, List<String> toList) {
        this.name = name;
        this.fromList = fromList;
        this.toList = toList;
    }

    @Override
    public void run() {
        String str = null;
```

# Thread

## ❖ DeadLock

@Override

```
public void run() {
```

```
    String str = null;
```

```
    try {
```

```
        System.out.printf("[%s] started.%n", name);
```

```
        Thread.sleep(500L);
```

```
        System.out.printf("[%s] attempt to lock fmList(%s).%n", name, fromList);
```

# Thread

## ❖ DeadLock

```
//fromList를 다른 Thread가 사용하지 못하도록 함
synchronized(fromList) {
    System.out.printf("[%s] fmList(%s) was locked.%n", name,
fromList);

    str = fromList.get(0);
    System.out.printf("[%s] %s <- fmList(%s)%n", name, str,
fromList);

    Thread.sleep(500L);
    System.out.printf("[%s] attempt to lock toList(%s).%n", name,
toList);

    //toList를 다른 Thread가 사용하지 못하도록 함
    synchronized(toList) {
        System.out.printf("[%s] toList(%s) was locked.%n",
name, toList);

        toList.add(str);
        System.out.printf("[%s] %s -> toList(%s)%n", name,
str, toList);
    }
}
```

# Thread

## ❖ DeadLock

```
        catch (InterruptedException e) {  
            e.printStackTrace();  
        } finally {  
            System.out.printf("[%s] finished.%n", name);  
        }  
    }  
}
```



# Thread

## ❖ DeadLock

```
import java.util.ArrayList;
import java.util.List;
public class DeadLockSample {
    public static void main(String... args) {
        List<String> list1 = new ArrayList<>();
        List<String> list2 = new ArrayList<>();
        list1.add("list1-1");
        list2.add("list2-1");
        new Thread(new ResourceLocker(" ThreadA", list1, list2)).start();
        new Thread(new ResourceLocker(" ThreadB", list2, list1)).start();
    }
}
```

# Thread

## ❖ 동시성 유틸리티

- ✓ jdk 1.5 버전부터는 Multi Thread 애플리케이션 개발을 용이하게 위한 클래스와 인터페이스로 구성된 동시성 유틸리티를 추가
- ✓ `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks` 패키지
- ✓ `java.lang` 패키지에 있는 Thread 관련 클래스들을 하위 레벨의 Threading API라고 하고 위 3개 패키지에 속한 클래스나 인터페이스는 상위 레벨의 Threading API
- ✓ 병렬 작업 처리가 많아지면 Thread 개수가 증가하게 되고 그에 따른 Thread 생성과 스케줄링으로 인하여 CPU가 바빠져서 실제 작업을 처리하는 시간보다 다른 부분을 처리하기 위한 시간이 늘어나는 경우가 발생하게 되서 애플리케이션의 성능이 저하
- ✓ 병렬 작업의 폭증으로 인한 Thread의 폭증을 막으려면 Thread Pool을 사용해야 함
- ✓ 작업 처리에 사용되는 Thread를 제한된 개수만큼 정해 놓고 작업 큐에 들어오는 작업을 하나씩 처리하는 방식으로 작업 처리가 끝난 Thread는 다시 작업 큐에서 새로운 작업을 가져와 처리하는 방식
- ✓ ThreadPool을 이용하게 되면 작업 처리 요청이 폭증하더라도 Thread의 전체 개수가 늘어나지는 않기 때문에 애플리케이션의 성능이 급격히 저하되지는 않음
- ✓ 자바 1.5에서는 이러한 ThreadPool을 생성하고 사용할 수 있도록 `java.util.concurrent` 패키지에서 `ExecutorService` 인터페이스와 `Executors` 클래스를 제공
- ✓ `java.util.concurrent.TimeUnit` : 시간 단위를 나타내는 열거형 상수로 `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, `SECONDS` 가 있음

# Thread

## ❖ 동시성 유틸리티

- ✓ `java.util.concurrent.Executors`: 자바 1.5에서 기본적으로 제공하는 `ExecutorService` 인터페이스를 구현한 객체를 생성해서 리턴하는 메서드를 제공하는 유틸리티 클래스
  - `ExecutorService newFixedThreadPool(int)`: 지정한 갯수만큼 `Thread`를 가질 수 있는 `ThreadPool`을 생성하는데 `Thread`가 아무 일도 하지 않더라도 제거되지 않음
  - `ExecutorService newCachedThreadPool()`: 재사용이 가능한 `ThreadPool`을 생성하는데 실행 가능한 `Thread`의 개수는 `int`의 최대값 만큼이며 `Thread`가 60초 동안 아무일도 하지 않으면 제거
  - `ExecutorService newSingleThreadExecutor()`: 단일 `Thread`만을 사용하는 `ExecutorService`를 생성
  - `ScheduledExecutorService newScheduledThreadPool(int)`: 스케줄링 가능한 `ThreadPool`을 생성
  - `ScheduledExecutorService newSingleThreadScheduledExecutor()`: 단일 `Thread`만을 사용하는 스케줄 가능한 `ExecutorService`를 생성
- ✓ `new ThreadPoolExecutor`(코어 `Thread` 개수, 최대 `Thread` 개수, 유희 시간, 시간단위, 작업 큐)를 이용해서 `ExecutorService` 객체 생성 가능
- ✓ `Runtime` 클래스의 `availableProcessors()`를 호출하면 현재 컴퓨터의 코어 수를 알 수 있음

# Thread

## ❖ 동시성 유틸리티

- ✓ `java.util.concurrent.ExecutorService` : `Executor`의 하위 인터페이스로서 생명주기 (`shutdown()`, `shutdownNow()`)를 관리할 수 있는 기능과 `Callable`이 구현된 객체를 사용할 수 있는 기능을 정의하고 있는 인터페이스
  - `shutdown()` : 중지(`shutdown`) 명령을 내리는 것으로 대기중인 작업은 실행되지만 새로운 작업은 추가되지 않음
  - `List<Runnable> shutdownNow()`: 현재 실행 중인 모든 작업 및 대기 중인 작업 모두를 중지시키는 메서드로 대기중인 작업 목록을 반환
  - `boolean isShutdown()`: `Executor`가 중지(`shutdown`)가 되었는지 여부를 판단해서 반환
  - `boolean isTerminated()`: 모든 작업이 종료되었는지 여부를 판단해서 리턴
  - `boolean awaitTermination(long, TimeUnit)`: 중지(`shutdown`) 명령을 내린 후, 지정한 시간 동안 모든 작업이 종료될 때까지 대기하는 메서드로 지정한 시간 이내에 작업이 종료되면 `true`, 아니면 `false`를 반환
  - `<T> Future<T> submit(Callable<T> task)`: 결과 값을 반환할 수 있는 `Callable` 작업을 실행

# Thread

## ❖ 동시성 유틸리티

### ✓ 작업 순서

- Executors의 static 메서드를 이용해서 ExecutorService 객체를 생성

```
ExecutorService executor =
```

```
    Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
```

```
ExecutorService executor = Executors.newCachedThreadPool();
```

```
ExecutorService executor = new ThreadPoolExecutor(코어 Thread 개수, 최대 Thread  
    개수, 유희 시간, 시간단위, 작업 큐)
```

- ExecutorService 객체의 submit 메서드에 Thread 객체를 추가

# Thread

## ❖ 동시성 유틸리티

```
public class MyThread implements Runnable {  
    private int i;  
  
    MyThread(int i) {  
        this.i = i;  
    }  
  
    public void run() {  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("나는 Thread:" + i);  
    }  
}
```

# Thread

## ❖ 동시성 유틸리티

```
public class MyThreadMain {  
    public static void main(String argv[]) throws Exception {  
        ExecutorService executor = Executors.newFixedThreadPool(3);  
        int i = 0;  
        while (!executor.isShutdown()) {  
            executor.submit(new MyThread(i++));  
            if (i > 10) {  
                try {  
                    executor.shutdown();  
                    boolean b = executor.awaitTermination(30,  
TimeUnit.SECONDS);  
                    System.out.println("남은 작업이 있다면 30초 동안 처  
리 한 후 종료합니다.");  
                    System.out.println("b:" + b);  
                    executor.shutdownNow();  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

# Thread

## ❖ java.util.concurrent.Callable

- ✓ 자바 1.5에서 제공하는 클래스로 결과를 리턴할 수 있는 call 이라는 메서드를 제공하는 Thread 인터페이스 – Thread의 run 메서드와 유사
- ✓ Runnable 인터페이스의 run 메서드는 결과를 리턴할 수 없지만 Callable 인터페이스의 call 메서드는 리턴 값이 있으므로 스레드의 수행 결과를 사용하는 것이 편리

V call() throws Exception

- ✓ Future<V>: Executors 에 submit 할 때 Callable 인터페이스를 implements 한 객체를 사용한 경우 결과를 받기 위한 클래스
  - V get(): Callable 작업의 실행이 완료될 때 까지 블로킹되며 완료되면 그 결과값을 리턴
  - V get(long timeout, TimeUnit unit): 지정한 시간 동안 작업의 실행 결과를 기다리고 지정한 시간 내에 수행이 완료되면 그 결과값을 리턴하지만 대기 시간이 초과되면 TimeoutException을 발생시킴
  - boolean cancel(boolean mayInterruptIfRunning): 작업을 취소
  - boolean isCancelled(): 작업이 정상적으로 완료되기 이전에 취소되었을 경우 true를 리턴
  - boolean isDone(): 작업이 완료되었다면 true를 리턴



# Thread

## ❖ Future

- ✓ 작업 결과가 아니고 작업이 완료될 때까지 기다렸다가 최종 결과를 얻는데 사용하는 클래스
- ✓ Future의 get()을 호출하면 Thread가 작업을 완료할 때까지 블로킹되었다가 작업을 완료하면 처리결과를 리턴
- ✓ 이 방식을 블로킹을 사용하는 작업 완료 통보 방식
- ✓ Thread 작업 처리 도중 예외가 발생하면 get()은 null을 리턴

# Thread

## ❖ 동시성 유틸리티

### ✓ Callable 과 Future

```
import java.util.concurrent.Callable;
public class CallThread implements Callable<Integer> {

    private int total;
    private int n;

    public CallThread(int n){
        this.n = n;
    }

    public Integer call() throws Exception {
        for (int i = 1; i <= n; i++) {
            total += i;
        }
        Thread.sleep(n);
        return total;
    }
}
```

# Thread

## ❖ 동시성 유틸리티

### ✓ Callable 과 Future

```
public class CallThreadMain {  
    public static void main(String[] args) {  
        ExecutorService executorService = Executors.newFixedThreadPool(3);  
        Future<Integer> f1 = executorService.submit(new CallThread(100));  
        Future<Integer> f2 = executorService.submit(new CallThread(20000));  
        Future<Integer> f3 = executorService.submit(new CallThread(17800));  
        Future<Integer> f4 = executorService.submit(new CallThread(2000));  
        Future<Integer> f5 = executorService.submit(new CallThread(12000));  
        Future<Integer> f6 = executorService.submit(new CallThread(80));  
    }  
}
```

# Thread

## ❖ 동시성 유틸리티

### ✓ Callable 과 Future

```
try {  
    System.out.printf("[%s] 1에서 100까지의 총 합은 %d입니다.%n",  
        "f1", f1.get());  
    System.out.printf("[%s] 1에서 20000까지의 총 합은 %d입니  
다.%n", "f2", f2.get());  
    System.out.printf("[%s] 1에서 17800까지의 총 합은 %d입니  
다.%n", "f3", f3.get());  
    System.out.printf("[%s] 1에서 2000까지의 총 합은 %d입니  
다.%n", "f4", f4.get());  
    System.out.printf("[%s] 1에서 12000까지의 총 합은 %d입니  
다.%n", "f5", f5.get());  
    System.out.printf("[%s] 1에서 80까지의 총 합은 %d입니다.%n",  
        "f6", f6.get());  
    executorService.shutdown();  
} catch (Exception e) {  
    System.out.println(e.getMessage());  
    e.printStackTrace();  
}  
}
```

# Thread

- ❖ `java.util.concurrent.locks` 패키지를 통해서 새로운 형태의 동기화 방식을 지원
  - ✓ `synchronized` 대신에 `ReentrantLock`이라는 클래스를 이용해서 Lock을 취득해서 Lock을 해제할 때 까지 Lock이 있는 블록에 접근이 안되도록 지원
  - ✓ `wait` 메서드 대신에 `await()` 메서드를 제공하고 `notify()` 대신에 `signal()` 메서드 `notifyAll()` 대신에 `signalAll()` 메서드를 제공하는데 메서드들은 `ReentrantLock` 클래스 기반의 동기화를 제공해야 하고 `ReentrantLock` 객체의 `newCondition()`이라는 메서드를 호출해서 얻어낸 `Condition` 객체를 이용해서 사용

# Thread

## ❖ 동기화

```
import java.util.concurrent.locks.ReentrantLock;

public class ShareDataEx implements Runnable {
    private int result;
    private int idx;
    static final ReentrantLock sLock = new ReentrantLock();
    public int getResult() {
        return result;
    }
    private void sum() {
        for (int i = 0; i < 10000; i++) {
            sLock.lock();
            idx++;
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
            }
            result += idx;
            sLock.unlock();
        }
    }
}
```

# Thread

## ❖ 동기화

```
@Override  
public void run() {  
    sum();  
}  
}
```



# Thread

## ❖ 동기화

```
public class ShareDataExMain {  
    public static void main(String[] args) {  
        ShareData share = new ShareData();  
        Thread th1 = new Thread(share);  
        Thread th2 = new Thread(share);  
  
        th1.start();  
        th2.start();  
  
        try{  
            Thread.sleep(30000);  
        }  
        catch(Exception e){}  
  
        System.out.println("res:" + share.getResult());  
    }  
}
```



# Thread

## ❖ 생산자와 소비자 문제

```
import java.util.*;
import java.util.concurrent.locks.*;

class ProductEx {
    ArrayList<Character> vec;
    static final ReentrantLock sLock = new ReentrantLock();
    Condition putCondition = sLock.newCondition();

    Product() {
        vec = new ArrayList<Character>();
    }
}
```

# Thread

## ❖ 생산자와 소비자 문제

```
put(char ch) {  
    sLock.lock();  
    while (vec.size() == 5) {  
        try {  
            putCondition.await();  
        } catch (InterruptedException e) {  
        }  
    }  
    System.out.println("생산자가 제품" + ch + "을(를) 생산했습니다.");  
    try {  
        Thread.sleep(1000);  
    } catch (Exception e) {  
    }  
    vec.add(new Character(ch));  
    System.out.print("창고에 제품" + ch + "가 입고 되었습니다.Wn");  
    System.out.print("재고수량:" + vec.size() + "Wn");  
    putCondition.signalAll();  
    sLock.unlock();  
}
```

# Thread

## ❖ 생산자와 소비자 문제

```
char get() {  
    sLock.lock();  
    while (vec.size() == 0) {  
        try {  
            putCondition.await();  
        } catch (InterruptedException E) {  
        }  
    }  
    try {  
        Thread.sleep(1000);  
    } catch (Exception e) {}  
    Character ch = (Character) vec.remove(0);  
    System.out.print("창고에서 제품" + ch + "가 출고 되었습니다" + "Wn");  
    System.out.print("재고수량:" + vec.size() + "Wn");  
    putCondition.signalAll();  
    sLock.unlock();  
    return ch.charValue();  
}  
}
```

# Thread

## ❖ java.util.concurrent.Semaphore

- ✓ 상호배제를 하면서 동시에 수행할 수 있는 Thread의 개수를 설정할 수 있는 클래스
- ✓ Executors는 상호배제와는 무관하게 동시에 수행할 수 있는 Thread 풀을 생성
- ✓ 생성자
  - Semaphore(int permits): 동시 수행 개수 지정
- ✓ 메서드
  - acquire(): 리소스를 확보하는 메서드 리소스에 빈자리가 생겼을 경우 바로 Thread가 acquire 메서드로부터 곧바로 돌아오게 되고 세마포어 내부에서는 리소스의 수가 하나 감소하며 리소스에 빈자리가 없을 경우 스레드는 acquire 메서드 내부에서 블록
  - release() - 리소스를 해제하는 메서드로 세마포어 내부에서는 리소스가 하나 증가하며 acquire 메서드 안에서 대기중인 Thread가 있으면 그 중 한 개가 깨어나 acquire 메서드로부터 돌아올 수 있음

# Thread

❖ java.util.concurrent.Semaphore

```
import java.util.concurrent.Semaphore;
```

```
class SemaphoreThread implements Runnable {  
    Semaphore sem;  
    String msg;
```

```
    SemaphoreThread(Semaphore s, String m) {  
        sem = s;  
        msg = m;  
    }
```

```
    public void run() {  
        try {  
            sem.acquire();  
            System.out.println(msg);  
            Thread.sleep(10000);  
        } catch (Exception exc) {  
            System.out.println("예외 발생");  
        }  
        sem.release();  
    }
```

```
}
```

# Thread

❖ java.util.concurrent.Semaphore

```
import java.util.concurrent.Semaphore;
```

```
public class SemaphoreMain {  
    public static void main(String args[]) throws Exception {  
        Semaphore sem = new Semaphore(3);  
        Thread thrdA = new Thread(new SemaphoreThread(sem, "Message 1"));  
        Thread thrdB = new Thread(new SemaphoreThread(sem, "Message 2"));  
        Thread thrdC = new Thread(new SemaphoreThread(sem, "Message 3"));  
        Thread thrdD = new Thread(new SemaphoreThread(sem, "Message 4"));  
  
        thrdA.start();  
        thrdB.start();  
        thrdC.start();  
        thrdD.start();  
  
    }  
}
```

# 타이머

## ❖ public class Timer extends Object

- ✓ 백그라운드 thread로 실행되는 태스크를 1회 또는 정기적으로 반복해 실행되도록 스케줄링해주는 클래스
- ✓ 생성자
  - Timer(): 새로운 타이머를 작성
  - Timer (boolean isDaemon): demon으로 실행되는 타이머
  - Timer (String name): 지정된 이름의 thread를 가지는 타이머
  - Timer (String name, boolean isDaemon): 지정된 이름의 thread를 가지는 새로운 타이머
- ✓ 메서드
  - void cancel(): 현재 스케줄 되고 있는 태스크를 파기해서 타이머를 종료
  - void schedule (TimerTask task, Date time): 지정된 시간으로 지정된 태스크가 실행되도록 스케줄
  - void schedule (TimerTask task, Date firstTime, long period): 지정된 태스크가 지정된 시간에 개시되어 period를 가지고 반복
  - void schedule (TimerTask task, long delay): 지정된 지연 후에 지정된 태스크가 실행되도록 스케줄
  - void schedule (TimerTask task, long delay, long period): 지정된 태스크가, 지정된 지연 후에 개시되어 period를 가지고 반복

# 타이머

## ❖ TimerTask 클래스

- ✓ public abstract class TimerTask extends Object implements Runnable
- ✓ Timer 에 해 1 회 또는 반복해 실행하도록 스케줄 되는 태스크
- ✓ 생성자
  - protected TimerTask () - 새로운 타이머 태스크가 작성
- ✓ 메서드
  - boolean cancel(): 이 타이머 태스크를 취소합니다.
  - abstract void run(): 이 타이머 태스크에 해 실행되는 액션
  - long scheduledExecutionTime(): 이 태스크의 최근 실행 시간을 리턴(이 메서드가 태스크의 실행 중에 불러 갔을 경우 리턴 값은 진행 중의 태스크 실행의 스케줄 된 실행 시간)



# 타이머

## ❖ Timer

```
import java.util.Timer;
import java.util.TimerTask;
class MyTask extends TimerTask {
    public void run() {
        System.out.println("타이머 호출");
    }
}
```



# 타이머

## ❖ Timer

```
import java.util.Timer;
```

```
public class TimerTest {  
    public static void main(String args[]) {  
        MyTask task = new MyTask();  
        Timer timer = new Timer();  
        timer.schedule(task, 1000, 500);  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException exc) {  
        }  
        timer.cancel();  
    }  
}
```

# 병렬 처리

## ❖ Fork & Join 프레임워크

- ✓ JDK 1.7에 추가된 API로 하나의 작업을 여러 개의 작은 단위로 나누어서 여러 개의 Thread로 처리 할 수 있도록 해주는 API
- ✓ 수행 할 작업의 리턴 여부에 따라 RecursiveAction(리턴 값이 없는 경우) 또는 RecursiveTask(리턴 값이 있는 경우)로 부터 상속받는 클래스를 만들어서 compute 메서드를 구현
- ✓ 위의 작업이 끝나면 ForkJoinPool 클래스를 이용해서 Thread 풀 객체를 생성
- ✓ ForkJoinPool 객체의 invoke 메서드에 RecursiveAction이나 RecursiveTask 객체를 생성해서 대입하면 작업을 나누어서 처리
- ✓ compute 메서드를 구현할 때는 작업을 어떤 식으로 나누어 처리할 것인지를 정의해야 함

# 병렬 처리

## ❖ 병렬 처리

```
import java.util.concurrent.RecursiveTask;

//start부터 n까지의 합계를 분할해서 실행하는 클래스
public class PartialTask extends RecursiveTask<Integer> {
    private static final long serialVersionUID = 1L;

    private int n;
    private int start;

    public PartialTask(int start, int n){
        this.start = start;
        this.n = n;
    }
}
```

# 병렬 처리

## ❖ 병렬 처리

```
private int sum(){
    int result = 0;
    for(int i=start; i<=n; i++){
        result += i;
    }
    return result;
}
```

```
@Override
protected Integer compute() {
    int size = n - start;
    if(size < 10)
        return sum();
    int half = (n+start)/2;
    PartialTask left = new PartialTask(start, half);
    PartialTask right = new PartialTask(half+1, n);
    left.fork();
    return right.compute() + left.join();
}
```

# 병렬 처리

```
import java.util.concurrent.ForkJoinPool;

public class PartialMain {
    public static void main(String[] args) {
        PartialTask task = new PartialTask(1, 100);
        ForkJoinPool pool = new ForkJoinPool();
        int r = pool.invoke(task);
        System.out.println(r);
    }
}
```

# 연습문제

1. Thread 클래스로 부터 상속받는 클래스를 생성(ThreadEx)해서 1부터 10까지의 숫자를 1초마다 출력하는 Thread 클래스를 생성하시오.
2. Runnable 인터페이스를 implements 한 클래스(RunnableImpl)를 생성해서 Thread Programming이라는 문자열을 1초마다 10번 출력하는 클래스를 생성하시오.
3. main 메서드를 생성하고 1, 2에서 만든 클래스의 인스턴스를 이용해서 Thread로 동작 시키시오.
4. Hello Anonymous를 1초마다 10번 출력하는 Thread를 Runnable 인터페이스를 구현한 Anonymous Class로 만들어서 동작시키는 코드를 위의 main 메서드에 작성하시오.