쓰레드 제어와 생명 주기1

소유자	종수 김
태그	

쓰레드 기본 정보

Thread 클래스는 쓰레드를 생성하고 관리하는 기능을 제공.

```
package thread.start.control;

import thread.start.HelloRunnable;

import static util.MyLoggerThread.log;

public class ThreadInfoMain {
    public static void main(String[] args) {
        // main 쓰레드
        Thread mainThread = Thread.currentThread();
        log("mainThread = " + mainThread);
        log("mainThread.threadId() = " + mainThread.threadId());
        log("mainThread.getName() = " + mainThread.getName());

        log("mainThread.getPriority() = " + mainThread.getPriority()); // 1~10
(기본값 5)

        log("mainThread.getThreadGroup() = " + mainThread.getThreadGroup
());

        log("mainThread.getState() = " + mainThread.getState());

        // myThread 스레드
        Thread myThread = new Thread(new HelloRunnable(), "myThread");
        log("myThread = " + myThread);
        log("myThread.threadId() = " + myThread.threadId());
        log("myThread.getName() = " + myThread.getName());
        log("myThread.getPriority() = " + myThread.getPriority());
```

```
        log("myThread.getThreadGroup() = " + myThread.getThreadGroup());
        log("myThread.getState() = " + myThread.getState());
    }
}

23:18:03.815 [     main] mainThread = Thread[#1,main,5,main]
23:18:03.818 [     main] mainThread.threadId() = 1
23:18:03.819 [     main] mainThread.getName() = main
23:18:03.821 [     main] mainThread.getPriority() = 5
23:18:03.821 [     main] mainThread.getThreadGroup() = java.lang.ThreadGr
oup[name=main,maxpri=10]
23:18:03.821 [     main] mainThread.getState() = RUNNABLE
23:18:03.821 [     main] myThread = Thread[#21,myThread,5,main]
23:18:03.821 [     main] myThread.threadId() = 21
23:18:03.822 [     main] myThread.getName() = myThread
23:18:03.822 [     main] myThread.getPriority() = 5
23:18:03.822 [     main] myThread.getThreadGroup() = java.lang.ThreadGro
up[name=main,maxpri=10]
23:18:03.822 [     main] myThread.getState() = NEW
```

2. 스레드 객체 정보

```
log("myThread = " + myThread);
```

- `myThread` 객체를 문자열로 변환하여 출력한다. `Thread` 클래스의 `toString()` 메서드는 스레드 ID, 스레드 이름, 우선순위, 스레드 그룹을 포함하는 문자열을 반환한다.
- `Thread[#21,myThread,5,main]`

3. 스레드 ID

```
log("myThread.threadId() = " + myThread.threadId());
```

- **threadId()**: 스레드의 고유 식별자를 반환하는 메서드이다. 이 ID는 JVM 내에서 각 스레드에 대해 유일하다. ID는 스레드가 생성될 때 할당되며, 직접 지정할 수 없다.

4. 스레드 이름

```
log("myThread.getName() = " + myThread.getName());
```

- **getName()**: 스레드의 이름을 반환하는 메서드이다. 생성자에서 `"myThread"` 라는 이름을 지정했기 때문에, 이 값이 반환된다. 참고로 스레드 ID는 중복되지 않지만, 스레드 이름은 중복될 수 있다.

5. 스레드 우선순위

```
log("myThread.getPriority() = " + myThread.getPriority());
```

- **getPriority()**: 스레드의 우선순위를 반환하는 메서드이다. 우선순위는 1 (가장 낮음)에서 10 (가장 높음)까지의 값으로 설정할 수 있으며, 기본값은 5이다. `setPriority()` 메서드를 사용해서 우선순위를 변경할 수 있다.
- 우선순위는 스레드 스케줄러가 어떤 스레드를 우선 실행할지 결정하는 데 사용된다. 하지만 실제 실행 순서는 JVM 구현과 운영체제에 따라 달라질 수 있다.

6. 스레드 그룹

```
log("myThread.getThreadGroup() = " + myThread.getThreadGroup());
```

- **getThreadGroup()**: 스레드가 속한 스레드 그룹을 반환하는 메서드이다. 스레드 그룹은 스레드를 그룹화하여 관리할 수 있는 기능을 제공한다. 기본적으로 모든 스레드는 부모 스레드와 동일한 스레드 그룹에 속하게 된다.
- 스레드 그룹은 여러 스레드를 하나의 그룹으로 묶어서 특정 작업(예: 일괄 종료, 우선순위 설정 등)을 수행할 수 있다.
- **부모 스레드(Parent Thread)**: 새로운 스레드를 생성하는 스레드를 의미한다. 스레드는 기본적으로 다른 스레드에 의해 생성된다. 이러한 생성 관계에서 새로 생성된 스레드는 생성한 스레드를 **부모**로 간주한다. 예를 들어 `myThread`는 `main` 스레드에 의해 생성되었으므로 `main` 스레드가 부모 스레드이다.
- `main` 스레드는 기본으로 제공되는 `main` 스레드 그룹에 소속되어 있다. 따라서 `myThread`도 부모 스레드인 `main` 스레드의 그룹인 `main` 스레드 그룹에 소속된다.
- **참고**: 스레드 그룹 기능은 직접적으로 잘 사용하지는 않기 때문에, 이런 것이 있구나 정도만 알고 넘어가자

7. 스레드 상태

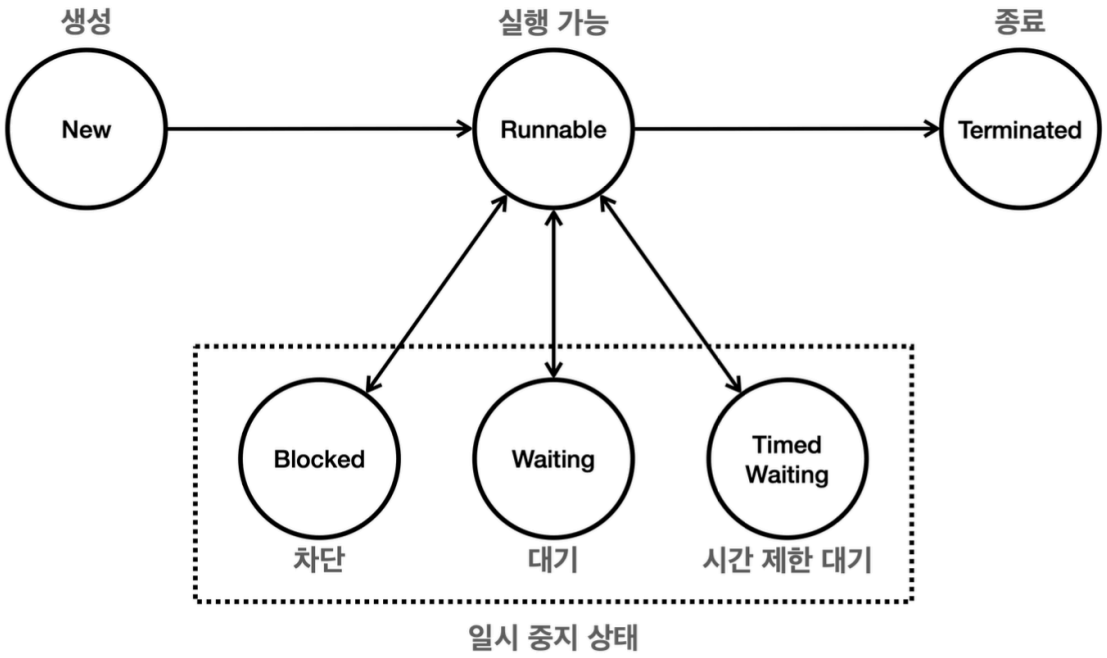
```
log("myThread.getState() = " + myThread.getState());
```

- **getState()**: 스레드의 현재 상태를 반환하는 메서드이다. 반환되는 값은 `Thread.State` 열거형에 정의된 상수 중 하나이다. 주요 상태는 다음과 같다.
  - **NEW**: 스레드가 아직 시작되지 않은 상태이다.
  - **RUNNABLE**: 스레드가 실행 중이거나 실행될 준비가 된 상태이다.
  - **BLOCKED**: 스레드가 동기화 락을 기다리는 상태이다.
  - **WAITING**: 스레드가 다른 스레드의 특정 작업이 완료되기를 기다리는 상태이다.
  - **TIMED_WAITING**: 일정 시간 동안 기다리는 상태이다.
  - **TERMINATED**: 스레드가 실행을 마친 상태이다.

쓰레드의 생명 주기 - 설명

쓰레드는 생성하고, 시작하고, 종료되는 생명 주기를 가짐.

그림 - 스레드 생명 주기



쓰레드의 상태

- New(새로운 상태) : 쓰레드가 생성되었으나 '아직 시작되지 않은 상태'
- Runnable(실행 가능 상태) : 쓰레드가 '실행 중이거나, 실행될 준비가 된 상태'
- 일시 중지 상태들 (Suspended States) :
  - Blocked (차단 상태) : 쓰레드가 동기화 락을 기다리는 상태
  - Waiting (대기 상태) : 쓰레드가 무기한으로 다른 쓰레드의 작업을 기다리는 상태
  - Timed Wating (시간 제한 대기 상태) : 쓰레드가 일정 시간 동안 다른 쓰레드의 작업을 기다리는 상태
- Terminated (종료 상태) : 쓰레드의 실행이 완료된 상태

> 일시 중지 상태들이란 없음.
> 단순히 설명을 위해 명명한 상태

1. New

- 쓰레드가 생성되고 아직 시작되지 않은 상태
- 이 상태에서는 Thread 객체가 생성되었지만, start() 메서드가 호출되지 않은 상태
- Ex) Thread thread = new Thread(runnable);

2. Runnable

- 쓰레드가 실행될 준비가 된 상태. 이 상태에서 쓰레드는 실제로 CPU에서 실행될 수 있음.
- start() 메서드가 호출되면 쓰레드는 이 상태로 들어감.
- 이 상태는 쓰레드가 실행될 준비가 되어 있음을 나타냄.
  즉, 실제로 CPU에서 실행될 수 있는 상태, 그러나 Runnable 상태에 있는 모든 쓰레드가 동시에 실행되는 것이 아니고, 운영체제의 스케줄러가 각 쓰레드에 CPU 시간을 할당하여 실행하기 때문에, Runnable 상태에 있는 쓰레드는 스케줄러의 실행 대기열에 있든, CPU에서 실제 실행되고 있던 모두 Runnable 인 것.

> CPU 입장에서는 '실제 실행하고 있는 상태, 스케쥴러에 등록되어 있는 상태' 엄청 빠르게 변경되며 실행되기에 Runnable은 '실행하고 있는 상태 + 실행될 준비가 된 상태' 인 것.
>
> **Runnable 상태의 쓰레드만 CPU의 실행 스케쥴러에 들어감.**

3. Blocked

- 쓰레드가 다른 쓰레드에 의해 동기화 락을 얻기 위해 기다리는 상태
- Ex) synchronized 블록에 진입하기 위해 락을 얻어야 하는 경우.

4. Waiting

- 쓰레드가 다른 쓰레드의 특정 작업이 완료되기를 무기한 기다리는 상태
- wait(), join() 메서드가 호출될 때 이 상태.
- 쓰레드는 다른 쓰레드가 notify() / notifyAll() 메서드를 호출하거나, join()이 완료될 때 까지 대기.

5. Timed Waiting

- 쓰레드가 특정 시간 동안 다른 쓰레드의 작업이 완료되기를 기다리는 상태
- sleep(), wait(), join() 메서드 호출 시
- 주어진 시간이 경과하거나, 다른 쓰레드가 해당 쓰레드를 깨우면 이 상태에서 벗어남.
- Ex) Thread.sleep(1000);

6. Terminated

- 쓰레드의 실행이 완료된 상태
- 쓰레드가 정상적으로 종료되거나, 예외가 발생하여 종료된 경우 이 상태
- 쓰레드는 한 번 종료되면 다시 시작할 수 없음.

자바 쓰레드의 상태 전이 과정

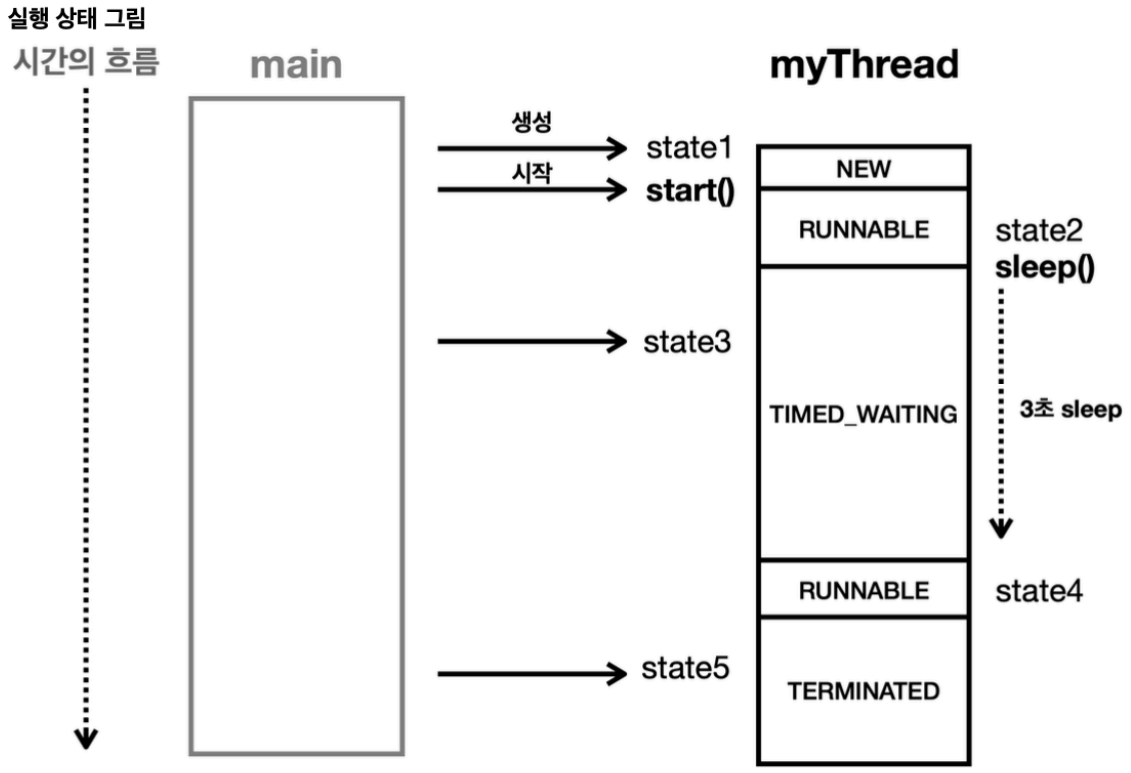
1. New → Runnable : start() 메서드가 호출되면, 쓰레드가 Runnable 상태로 전이
2. Runnable → Blocked / Waiting / Timed Waiting : 쓰레드가 락을 얻지 못하거나, wait(), sleep() 등의 메서드를 호출할 때 전이
3. Blocked / Waiting / Timed Waiting → Runnable : 쓰레드가 락을 얻거나, 기다림이 완료 되면 Runnable 상태로 전이
4. Runnable → Terminated : 쓰레드의 Run() 메서드가 완료되면 쓰레드는 Terminated

쓰레드의 생명 주기 - 코드

쓰레드가 생성, 실행, 대기 및 종료 상태로 변경될 때 마다 해당 상태를 로그로 출력

```
package thread.start.control;

import static util.MyLoggerThread.log;

public class ThreadStateMain {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new MyRunnable(), "myThread");
        log("myThread.state1 = " + thread.getState()); // NEW
        log("myThread.start()");
        thread.start();
        Thread.sleep(1000);
        log("myThread.state3 = " + thread.getState()); // TIME_WAITING
        Thread.sleep(4000);
        log("myThread.state5 = " + thread.getState()); // TERMINATED

    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {

            try {
                log("start");
                log("myThread.state2 = " + Thread.currentThread().getState()); //
```

```
RUNNABLE
            log("sleep() start");
            Thread.sleep(3000);
            log("sleep() end");
            log("myThread.state4 = " + Thread.currentThread().getState()); //
RUNNABLE
            log("end");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
  }
}
```



- `main` 스레드의 상태는 생략했다. 여기서는 `myThread` 스레드의 상태에 집중하자

1. state1 = NEW

- main 쓰레드를 통해 myThread 객체를 생성.
- 쓰레드 객체만 생성하고 아직 start()를 호출하지 않았기에 NEW 상태.

2. state2 = RUNNABLE

- myThread.start()를 호출해서 myThread를 실행 상태로 만듦.
- 따라서, RUNNABLE 상태
- 참고로, 실행 상태가 너무 빨리 지나가기에 main 쓰레드에서 myThread의 상태를 확인하긴 어려움.
  대신 자기 자신인 myThread 에서 실행 중신 자신의 상태를 확인.

3. state3 = TIMED_WAITING

- Thread.sleep(3000) 해당 코드를 호출한 쓰레드는 3000ms간 대기, myThread가 해당 코드를 호출 했으므로 3초간 대기하면서 TIMED_WATING 상태로 변함.

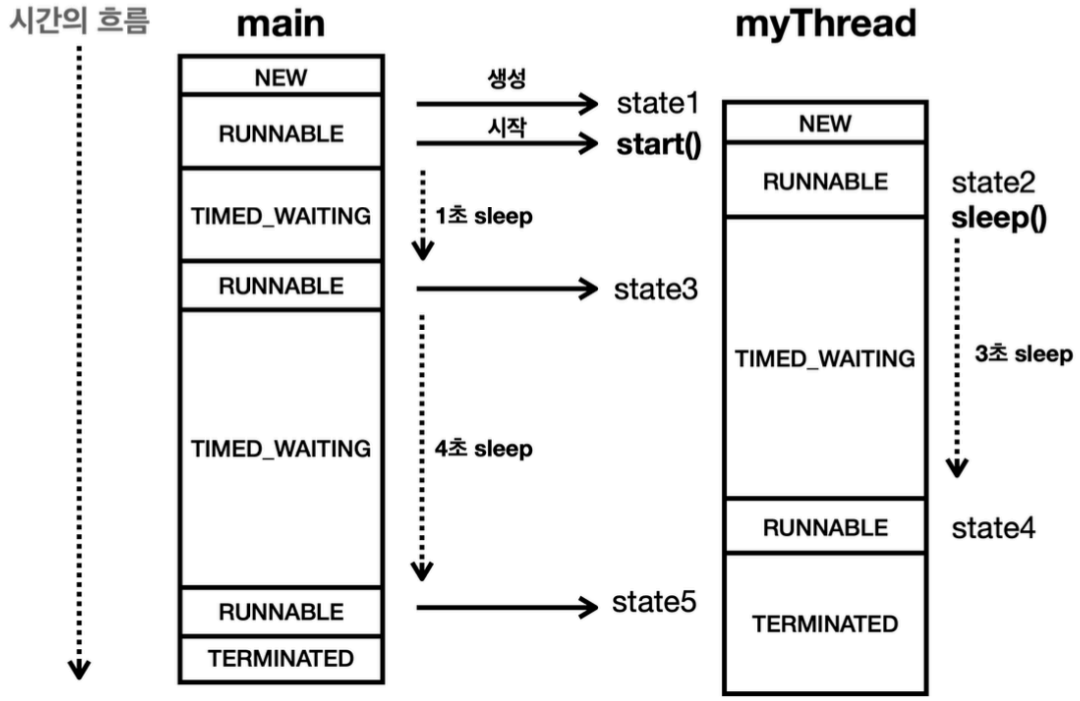
4. state4 = RUNNABLE

- myThread는 3초의 시간 대기 후 TIMED_WAITING 상태에서 빠져나와 다시 실행될 수 있는 Runnable 상태로 전이

5. state5 = TERMINATED

- myThread가 run() 메서드를 실행 종료하고 나면 TERMINATED 상태가 됨.
- myThread 입장에서 run() 이 스택에 남은 마지막 메서드인데, run() 까지 실행되고 나면 스택이 완전히 비워짐. 이렇게 스택이 비워지면 해당 스택을 사용하는 쓰레드도 종료.

실행 상태 그림 - main 스레드 포함



- `main` 스레드의 상태까지 포함한 전체 그림이다. 참고로만 봐두자.

체크 예외 재정의

Runnable 인터페이스의 run() 메서드를 구현할 때, InterruptedException 체크 예외를 밖으로 던질 수 없는 이유

Runnable Interface

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

자바에서 메서드를 재정의 할 때, 재정의 메서드가 지켜야 할 예외와 관련된 규칙

- 체크 예외
    - **부모 메서드가 체크 예외를 던지지 않는 경우, 재정의 된 자식 메서드도 체크 예외를 던질 수 없음.**
    - 자식 메서드는 부모 메서드가 던질 수 있는 체크 예외의 하위 타입만 던질 수 없음.
- 언체크(런타임) 예외
    - 예외 처리를 강제하지 않으므로 상관없이 던질 수 있음.

즉, Runnable 인터페이스의 run() 메서드는 아무런 체크예외를 던지지 않기 때문에, 이를 오버라이딩 하는 자식 메서드 내에서도 체크 예외를 밖으로 던질 수 없음.

```
package thread.start.control;

public class CheckedExceptionMain {
    public static void main(String[] args) throws Exception{
        throw new Exception();
    }

    static class CheckRunnable implements Runnable {
        @Override
        public void run() throws Exception {
            throw new Exception(); // Compile Error !
        }
    }
}
```

```
// 위 이유로 해당 Exception도 던질 수 없는 것.
static class MyRunnable implements Runnable {
    public void run() throws InterruptedException {
        Thread.sleep(3000);
    }
}
```

Why ?

부모 클래스의 메서드를 호출하는 클라이언트 코드는 부모 메서드가 던지는 특정 예외만을 처리하도록 작성된다.

자식 클래스가 더 넓은 범위의 예외를 던지면 **해당 코드는 모든 예외를 제대로 처리하지 못할 수 있다.**

이는 예외 처리의 일관성을 해치고 예상치 못한 런타임 오류를 초래할 수 있는 것.

```
class Parent {
    void method() throws InterruptedException {
// ...
} }
 class Child extends Parent {
    @Override
    void method() throws Exception {
        // ...
} }
 public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        try {
            p.method();
        } catch (InterruptedException e) {
// InterruptedException 처리 }
} }
```

- 자바 컴파일러는 `Parent p` 의 `method()` 를 호출한 것으로 인지한다.
- `Parent p` 는 `InterruptedException` 를 반환하는데, 그 자식이 전혀 다른 예외를 반환한다면 클라이언트는 해당 예외를 잡을 수 없다.

- 이것은 확실하게 모든 예외를 체크하는 체크 예외의 규약에 맞지 않는다.
- 따라서 자바에서 체크 예외의 메서드 재정의는 다음과 같은 규칙을 가진다
- 체크 예외는 컴파일 타임의 안전성을 확보하기 위함인데 아래와 같은 경우 컴파일 타임에서의 안전성을 보장하지 못함.

```
Parent p = new Child();
Parent : FileNotFoundException
Child : IOException

이런 구조라면, p.method() 실행 시 try-catch를 FileNotFoundException으로 할지, IOException 으로 할지 모르니까 ? << 컴파일 타임에 코드에 대한 안정성이 확보되지 않음
```

안전한 예외 처리

체크 예외를 run() 메서드에서 던질 수 없도록 강제함으로써, 개발자는 반드시 체크 예외를 try-catch 블록 내에서 처리하게 됨.

이는 예외 발생 시 예외가 적절히 처리되지 않아서 프로그램이 비정상 종료되는 것을 막을 수 있음.

특히 멀티스레딩 환경에서는 예외 처리를 강제함으로써 쓰레드의 안정성과 일관성을 유지할 수 있음.

> 최근에는 체크 예외보다는 언체크(런타임) 예외를 선호.
>
> 1. **코드 가독성·유지보수성 확보**
>   - 모든 메서드에서 try-catch 강제 → 오히려 코드 난잡
>   - 공통단에서 한 번만 처리하면 됨 → 비즈니스 로직 집중 가능
> 2. **글로벌 예외 처리 가능**
>   - 스프링: @ControllerAdvice, @ExceptionHandler로 **전역에서 통합 처리**
>   - 개별 서비스/컨트롤러에서 반복적 처리 제거 가능
> 3. **설계 철학: 비즈니스 로직 중심**
>   - 기술적 예외(DAO, IO 등)는 런타임 예외로 처리 → **비즈니스 로직 방해 최소화**
>   - 체크드 예외처럼 “꼭 잡게 강제”하면 코드 흐름이 방해됨

Join - 시작

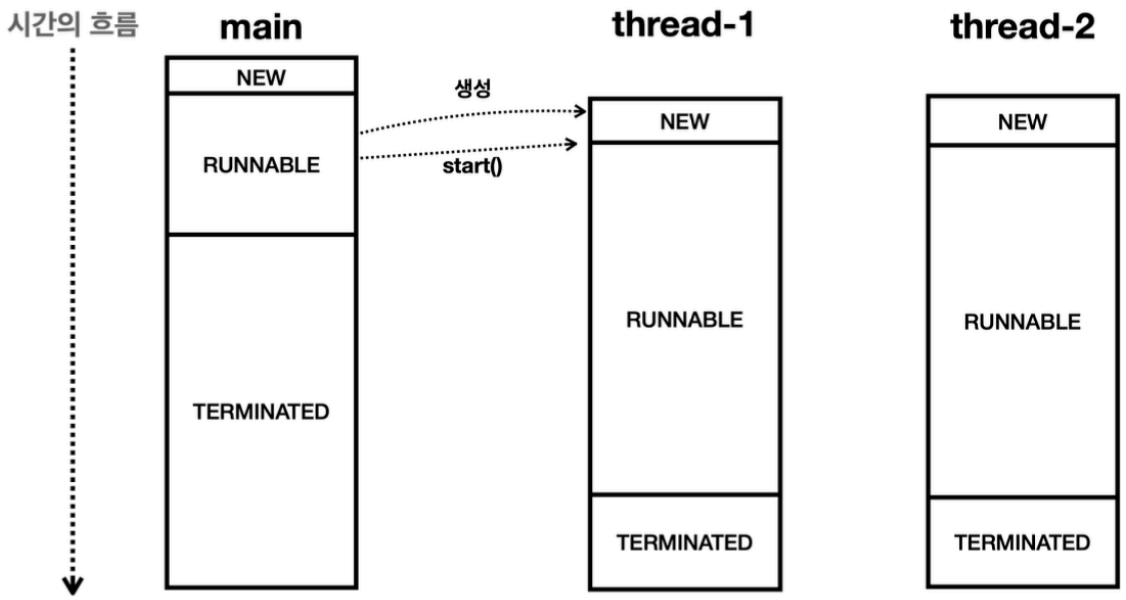
Waiting (대기 상태)

- 쓰레드가 다른 쓰레드의 특정 작업이 완료되기를 무기한 기다리는 상태.

TimedWating은 특정 시간만큼만, Waiting은 무기한 대기

```
package thread.start.control.join;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class JoinMainV0 {
    public static void main(String[] args) {
        log("start");
        Thread thread1 = new Thread(new Job(), "thread-1");
        Thread thread2 = new Thread(new Job(), "thread-2");

        thread1.start();
        thread2.start();
        log("end");
    }
    static class Job implements Runnable {
        @Override
        public void run() {
            log("작업 시작");
            sleep(2000);
            log("작업 완료");
        }
    }
}

19:36:20.989 [     main] start
19:36:20.990 [ thread-1] 작업 시작
19:36:20.990 [ thread-2] 작업 시작
19:36:20.990 [     main] end
19:36:22.995 [ thread-2] 작업 완료
19:36:22.996 [ thread-1] 작업 완료
```

참고: 스레드의 실행 순서는 보장되지 않기 때문에, 실행 결과는 약간 다를 수 있다.



- 그림에서는 생략했지만, `thread-2`도 `main` 스레드가 생성하고 `start()`를 호출해서 실행한다.

sleep() 을 사용해서 2초간 대기 할 경우, TimedWaiting 상태이지만, 작업을 한다는 가정으로 Runnable

- main 쓰레드는 thread-1, thread-2를 실행하고 바로 자신의 다음 코드를 실행함.
- 핵심은, thread-1, thread-2가 끝날 때 까지 기다리지 않는다는 것.
- main 쓰레드는 단지 start()를 호출해서 다른 쓰레드를 실행만 하고, 바로 자신의 다음 코드를 실행.

만약, thread-1, thread-2 가 종료된 이후 main 쓰레드를 가장 마지막에 종료하려면 ?

즉, main 쓰레드가 thread-1, thread-2에 각각 어떤 작업을 지시하고, 그 결과를 받아서 처리하고 싶다면 ?

Join - 필요한 상황

`1~100`까지 더하는 간단한 코드를 작성해보자.

```
int sum = 0;
for (int i = 1; i <= 100; i++) {
    sum += i;
}
```

이 코드는 스레드를 하나만 사용하기 때문에 CPU 코어도 하나만 사용할 수 있다. CPU 코어를 더 효율적으로 사용하려면 여러 스레드로 나누어 계산하면 된다.

`1 ~ 100`까지 더한 결과는 `5050`이다. 이 연산은 다음과 같이 둘로 나눌 수 있다.

- `1 ~ 50`까지 더하기 = `1275`
- `51 ~ 100`까지 더하기 = `3775`

두 계산 결과를 합하면 `5050`이 나온다.

`main` 스레드가 `1 ~ 100`으로 더하는 작업을 `thread-1`, `thread-2`에 각각 작업을 나누어 지시하면 CPU 코어를 더 효율적으로 활용할 수 있다. CPU 코어가 2개라면 이론적으로 연산 속도가 2배 빨라진다.

- `thread-1`: `1 ~ 50`까지 더하기
- `thread-2`: `51 ~ 100`까지 더하기
- `main`: 두 스레드의 계산 결과를 받아서 합치기(이건 간단한 연산 한 번이니 속도 계산에서 제외하자)

두 수 합치기

```
package thread.start.control.join;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class JoinMainV1 {
    public static void main(String[] args) {
        log("start");
        SumTask task1 = new SumTask(1, 50);
        SumTask task2 = new SumTask(51, 100);

        Thread thread1 = new Thread(task1, "thread-1");
        Thread thread2 = new Thread(task2, "thread-2");

        thread1.start();
        thread2.start();
```

```
        log("task1.result = " + task1.result);
        log("task2.result = " + task2.result);

        int sumAll = task1.result + task2.result;
        System.out.println(sumAll);
        log("end");
    }

    static class SumTask implements Runnable {
        /*
            int 이므로 초기화 안해도 0
         */
        int startValue;
        int endValue;
        int result = 0;


        public SumTask(int startValue, int endValue) {
            this.startValue = startValue;
            this.endValue = endValue;
        }

        @Override
        public void run() {
            log("작업 시작");
            sleep(2000);
            int sum = 0;
            for(int i = startValue; i <= endValue; i++) {
                sum += i;
            }
            result = sum;
            log("작업 완료 = " + result);
        }
    }
}

20:05:19.055 [     main] start
20:05:19.058 [ thread-1] 작업 시작
```

```
20:05:19.058 [ thread-2] 작업 시작
20:05:19.060 [     main] task1.result = 0
20:05:19.060 [     main] task2.result = 0
0
20:05:19.060 [     main] end
20:05:21.060 [ thread-1] 작업 완료 = 1275
20:05:21.062 [ thread-2] 작업 완료 = 3775
```

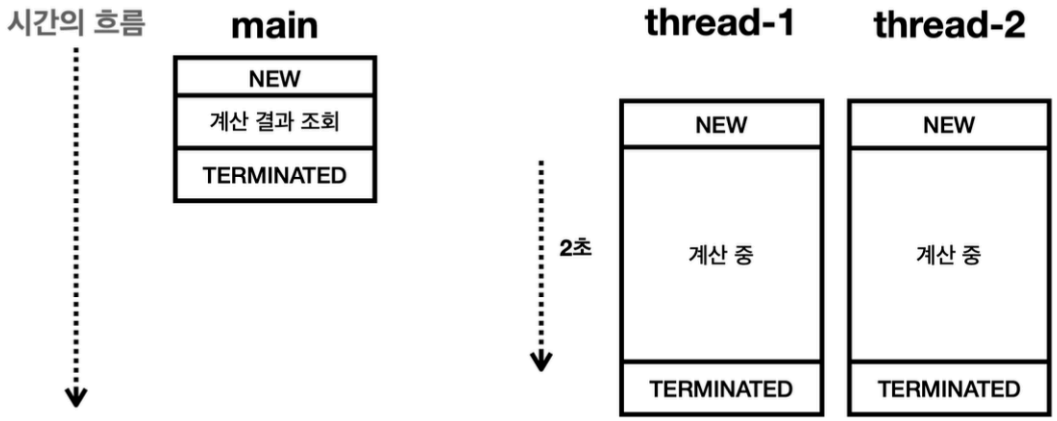
SumTask는 계산의 시작 값과, 마지막 값을 가짐. 그리고 계산이 끝나면 그 결과를 result에 담아둠.

thread-1은 task1 인스턴스의 run()을 실행하고, thread-2sms task2 인스턴스의 run()을 실행함.

결과는 0

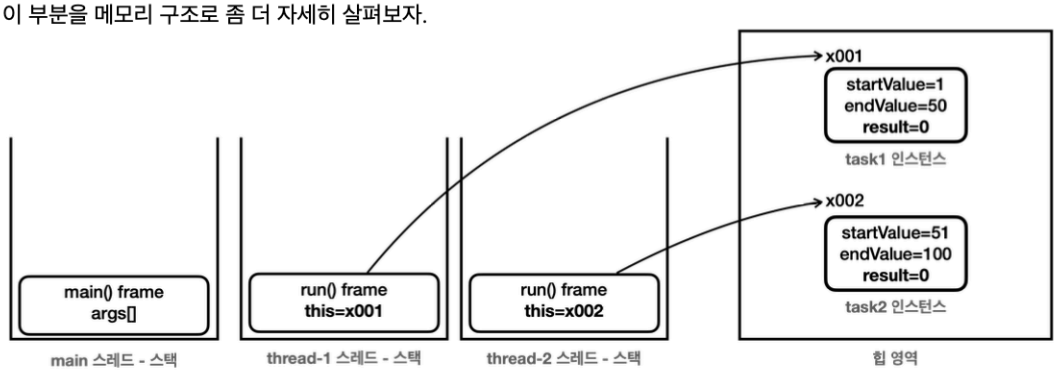
초기 할당

실행 결과 분석

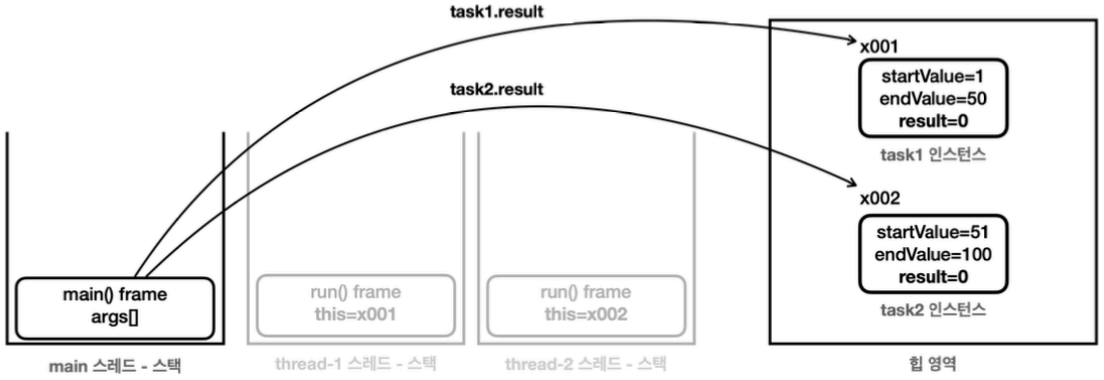


`main` 스레드는 `thread-1`, `thread2`에 작업을 지시하고, `thread-1`, `thread2`가 계산을 완료하기도 전에 먼저 계산 결과를 조회했다. 참고로 `thread-1`, `thread-2`가 계산을 완료하는데는 2초 정도의 시간이 걸린다. 따라서 결과가 `task1 + task2 = 0`으로 출력된다.

이 부분을 메모리 구조로 좀 더 자세히 살펴보자.



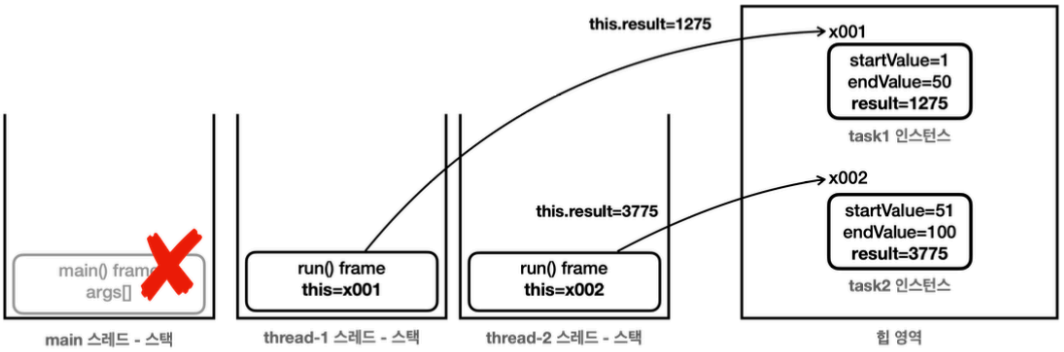
main 수행



- main 쓰레드는 두 쓰레드를 시작한 다음에 바로 task1.result, task2.result를 통해 인스턴스에 있는 값을 조회함.

- main 쓰레드가 실행한 start() 메서드는 쓰레드의 실행이 끝나는 것을 기다리지 않고, '실행만 해둔 후 자신의 다음 코드를 실행'

main 종료



- 2초가 지난 이후 thread-1, thread-2는 계산을 완료함.
- 이 때, main 쓰레드는 이미 자신의 코드를 모두 실행하고 종료된 상태.

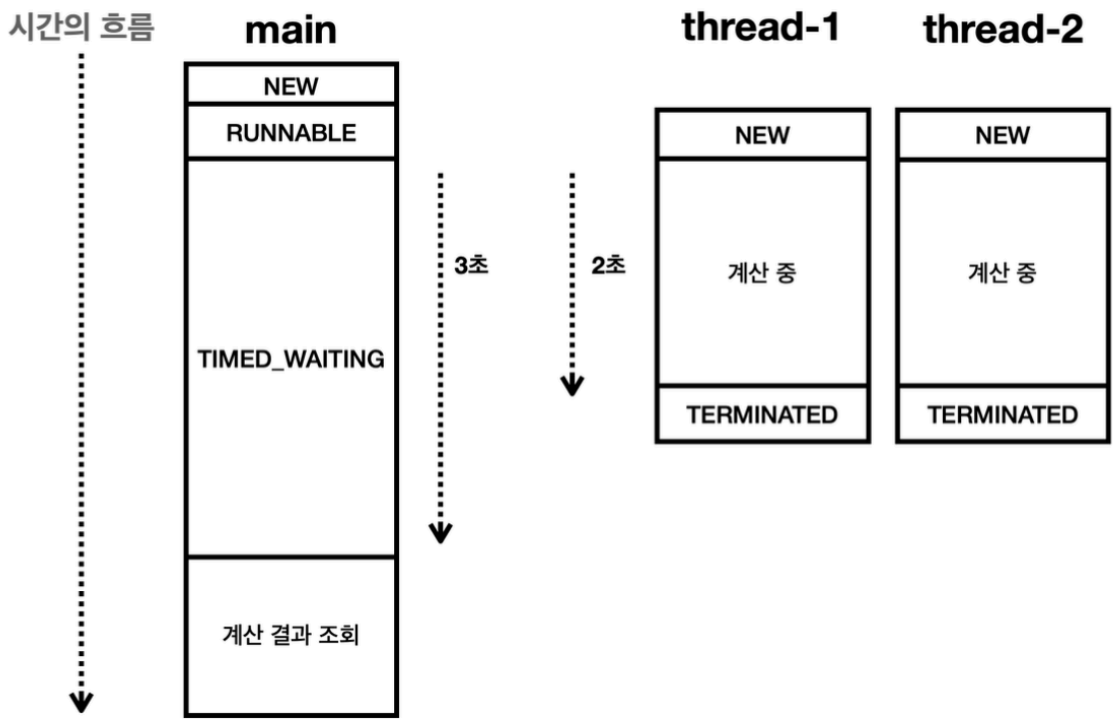
> this
>
> 어떤 메서드를 호출하는 것은, 정확히는 특정 스레드가 어떤 메서드를 호출하는 것이다.
>
> 스레드는 메서드의 호출을 관리하기 위해 메서드 단위로 스택 프레임을 만들고 해당 스택 프레임을 스택위에 쌓아 올린다.
> 이때 인스턴스의 메서드를 호출하면, 어떤 인스턴스의 메서드를 호출했는지 기억하기 위해, 해당 인스턴스의 참조값을스택 프레임 내부에 저장해둔다. 이것이 바로 우리가 자주 사용하던 `this` 이다.
>
> 특정 메서드 안에서 `this` 를 호출하면 바로 스택프레임 안에 있는 `this` 값을 불러서 사용하게 된다.
> 그림을 보면 스택 프레임 안에 있는 `this` 를 확인할 수 있다. 이렇게 `this` 가 있기 때문에 `thread-1` ,`thread-2`는 자신의 인스턴스를 구분해서 사용할 수 있다. 예를 들어서 필드에 접근할 때 `this` 를 생략하면 자동으로 `this` 를참고해서 필드에 접근한다.
>
> 정리하면 `this` 는 호출된 인스턴스 메서드가 소속된 객체를 가리키는 참조이며, 이것이 스택 프레임 내부에 저장되어있다.
>
> 즉, this는 스택 프레임이 가지고 있는 인스턴스의 참조 값

Join - Sleep 사용

특정 쓰레드를 기다리게 하는 가장 간단한 방법 sleep()

```
package thread.start.control.join;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class JoinMainV2 {
    public static void main(String[] args) {
        log("start");
        SumTask task1 = new SumTask(1, 50);
        SumTask task2 = new SumTask(51, 100);

        Thread thread1 = new Thread(task1, "thread-1");
        Thread thread2 = new Thread(task2, "thread-2");

        thread1.start();
        thread2.start();

        log("task1.result = " + task1.result);
        log("task2.result = " + task2.result);


        // 정확한 타이밍을 맞추어 기다리기 어려움.
        log("main Thread Sleep()");
        sleep(3000);
        log("main Thread 기상");

        int sumAll = task1.result + task2.result;
        System.out.println(sumAll);
        log("end");
    }

    static class SumTask implements Runnable {
        /*
            int 이므로 초기화 안해도 0
         */
        int startValue;
```

```
        int endValue;
        int result = 0;


        public SumTask(int startValue, int endValue) {
            this.startValue = startValue;
            this.endValue = endValue;
        }

        @Override
        public void run() {
            log("작업 시작");
            sleep(2000);
            int sum = 0;
            for(int i = startValue; i <= endValue; i++) {
                sum += i;
            }
            result = sum;
            log("작업 완료 = " + result);
        }
    }
}

20:17:10.685 [     main] start
20:17:10.688 [ thread-2] 작업 시작
20:17:10.688 [ thread-1] 작업 시작
20:17:10.690 [     main] task1.result = 0
20:17:10.690 [     main] task2.result = 0
20:17:10.690 [     main] main Thread Sleep()
20:17:12.691 [ thread-2] 작업 완료 = 3775
20:17:12.691 [ thread-1] 작업 완료 = 1275
20:17:13.696 [     main] main Thread 기상
5050
20:17:13.696 [     main] end
```



- `thread-1`, `thread-2`는 계산에 2초 정도의 시간이 걸린다. 우리는 이 부분을 알고 있어서 `main` 스레드가 약 3초 후에 계산 결과를 조회하도록 했다. 따라서 계산된 결과를 받아서 출력할 수 있다.

- 하지만, 이렇게 sleep()을 사용하여 무작정 기다리는 방법은, 대기 시간 손해 및 thread-1, thread-2의 수행시간이 달라지는 경우 정확한 타이밍을 맞출 수 없음.

thread-1, thread-2가 계산을 끝내고 종료될 때 까지 main 쓰레드가 기다리는 방법

Ex) Main 쓰레드가 반복문을 사용해서 thread-1, thread-2의 상태가 TERMINATED가 될 때 까지 계속 확인하는 방법.

```
while(thread.getState() != TERMINATED) { //스레드의 상태가 종료될 때 까지 계속 반복
}
//계산 결과 출력
```

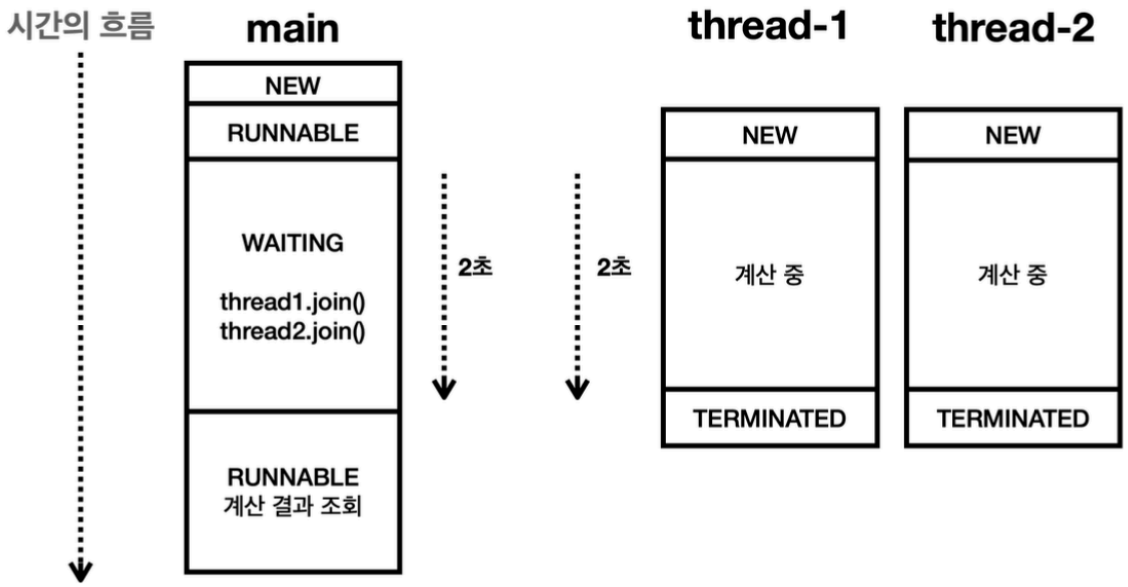
- 매 번 멀티 쓰레드 작업마다 이렇게 놓을 순 없음.

Join - join() 사용

```
package thread.start.control.join;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;
```

```
public class JoinMainV3 {
    public static void main(String[] args) throws InterruptedException {
        log("start");
        SumTask task1 = new SumTask(1, 50);
        SumTask task2 = new SumTask(51, 100);

        Thread thread1 = new Thread(task1, "thread-1");
        Thread thread2 = new Thread(task2, "thread-2");

        thread1.start();
        thread2.start();

        // Thread가 종료될 때 까지 대기
        log("join() - main 쓰레드가 thread-1, thread-2 종료까지 대기");
        thread1.join();
        thread2.join();
        log("main 쓰레드 대기 완료");

        log("task1.result = " + task1.result);
        log("task2.result = " + task2.result);

        int sumAll = task1.result + task2.result;
        System.out.println(sumAll);
        log("end");
    }

    static class SumTask implements Runnable {
        /*
            int 이므로 초기화 안해도 0
         */
        int startValue;
        int endValue;
        int result = 0;


        public SumTask(int startValue, int endValue) {
            this.startValue = startValue;
```

```
            this.endValue = endValue;
        }

        @Override
        public void run() {
            log("작업 시작");
            sleep(2000);
            int sum = 0;
            for(int i = startValue; i <= endValue; i++) {
                sum += i;
            }
            result = sum;
            log("작업 완료 = " + result);
        }
    }
}

20:24:01.221 [     main] start
20:24:01.223 [ thread-1] 작업 시작
20:24:01.224 [     main] join() - main 쓰레드가 thread-1, thread-2 종료까지 대
기
20:24:01.224 [ thread-2] 작업 시작
20:24:03.239 [ thread-1] 작업 완료 = 1275
20:24:03.239 [ thread-2] 작업 완료 = 3775
20:24:03.240 [     main] main 쓰레드 대기 완료
20:24:03.241 [     main] task1.result = 1275
20:24:03.241 [     main] task2.result = 3775
5050
20:24:03.242 [     main] end
```



`main` 스레드에서 다음 코드를 실행하게 되면 `main` 스레드는 `thread-1`, `thread-2`가 종료될 때 까지 기다린다.
이때 `main` 스레드는 `WAITING` 상태가 된다.

```
thread1.join();
thread2.join();
```

- thread-1이 아직 종료되지 않았다면 main 쓰레드는 thread1.join() 코드 안에서 더는 진행하지 않고, 멈추어 기다림.
- 이후에 thread-1이 종료되면, main 쓰레드는 Runnable 상태가 되고 다음 코드로 이동.
- 이 때 thread-2가 아직 종료되지 않았다면 main 쓰레드는 thread2.join() 코드 안에서 진행하지 않고 멈추어 기다림.
- 이후에 thread-2가 종료되면 main 쓰레드는 RUNNABLE 상태가 되고 다음 코드로 이동

WAITING

- 쓰레드가 다른 쓰레드의 특정 작업이 완료되기를 무기한 기다리는 상태.
- join()을 호출하는 쓰레드는 대상 쓰레드가 TERMINATED 상태가 될 때 까지 대기
- 대상 쓰레드가 TERMINATED가 되면 호출 쓰레드는 다시 RUNNABLE 상태가 되면서 다음 코드를 수행

이렇게 특정 쓰레드가 완료될 때 까지 기다려야하는 상황이라면 join()을 사용.

하지만, 다른 쓰레드가 완료될 때 까지 무기한 기다려야함. 만약 다른 쓰레드의 작업을 일정 시간 동안만 기다리게하고 싶다면 ?

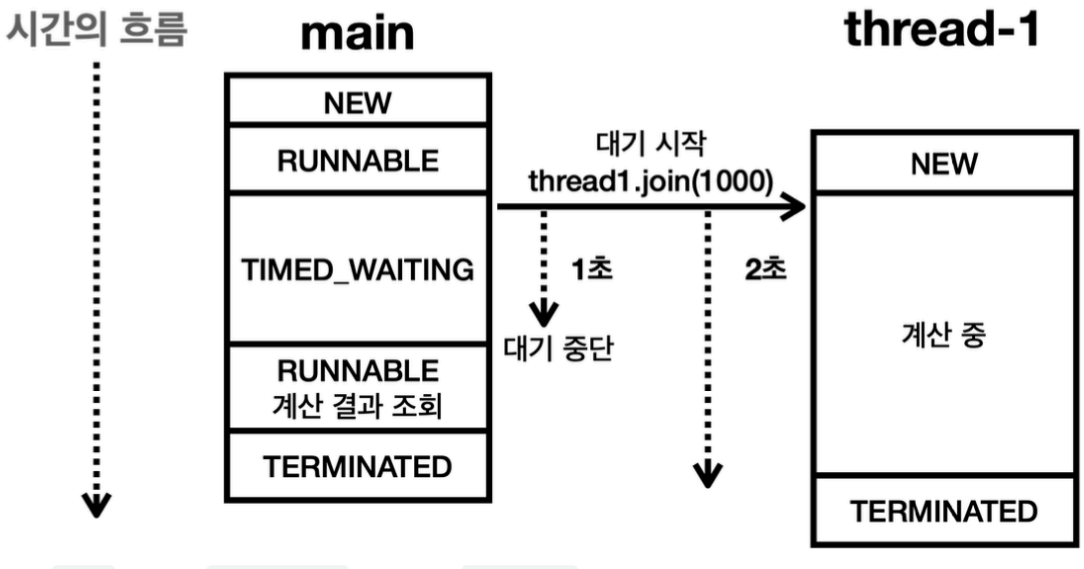
Join - 특정 시간 만큼만 대기

join()은 두 가지 메서드가 있음.

1. join() : 호출 쓰레드는 대상 쓰레드가 완료될 때 까지 무기한 대기
2. join(ms) : 호출 쓰레드는 특정 시간 만큼만 대기, 호출 쓰레드는 지정한 시간이 지나면 RUNNABLE 상태로 돌아간다.

```
package thread.start.control.join;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class JoinMainV4 {
    public static void main(String[] args) throws InterruptedException {
        log("start");
        SumTask task1 = new SumTask(1, 50);

        Thread thread1 = new Thread(task1, "thread-1");

        thread1.start();

        // Thread가 종료될 때 까지 대기
        log("join() - main 쓰레드가 thread1 종료까지 1초 대기");
        thread1.join(1000);
        log("main 쓰레드 대기 완료");

        log("task1.result = " + task1.result);
        log("end");
    }

    static class SumTask implements Runnable {
        /*
            int 이므로 초기화 안해도 0
         */
        int startValue;
```

```
        int endValue;
        int result = 0;


        public SumTask(int startValue, int endValue) {
            this.startValue = startValue;
            this.endValue = endValue;
        }

        @Override
        public void run() {
            log("작업 시작");
            sleep(2000);
            int sum = 0;
            for(int i = startValue; i <= endValue; i++) {
                sum += i;
            }
            result = sum;
            log("작업 완료 = " + result);
        }
    }
}

20:30:04.570 [     main] start
20:30:04.573 [     main] join() - main 쓰레드가 thread1 종료까지 1초 대기
20:30:04.573 [ thread-1] 작업 시작
20:30:05.578 [     main] main 쓰레드 대기 완료
20:30:05.588 [     main] task1.result = 0
20:30:05.588 [     main] end
20:30:06.580 [ thread-1] 작업 완료 = 1275
```



- main 쓰레드는 join(1000)을 사용해서 thread-1을 1초간 기다림.
  - 이 때 main 쓰레드의 상태는 WAITING이 아니라, TIMED_WATING
  - 보통 무기한 대기하면 WAITING, 특정 시간 만큼만 대기하면 TIMED_WAITING