3. 스레드 제어와 생명 주기1

#1.인강/0.자바/5.자바-고급1편

스레드 기본 정보

`Thread` 클래스는 스레드를 생성하고 관리하는 기능을 제공한다.
`Thread` 클래스가 제공하는 정보들을 확인해보자.
하나는 기본으로 제공되는 `main` 스레드의 정보를, 하나는 직접 만든 `myThread` 스레드의 정보를 출력해보자.

```
package thread.control;

import thread.start.HelloRunnable;

import static util.MyLogger.log;

public class ThreadInfoMain {

    public static void main(String[] args) {
        // main 스레드
        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());
    }
}
```

실행 결과

```
//main 스레드 출력
09:55:58.709 [     main] mainThread = Thread[#1,main,5,main]
09:55:58.713 [     main] mainThread.threadId() = 1
09:55:58.713 [     main] mainThread.getName() = main
09:55:58.716 [     main] mainThread.getPriority() = 5
09:55:58.716 [     main] mainThread.getThreadGroup() =
java.lang.ThreadGroup[name=main,maxpri=10]
09:55:58.716 [     main] mainThread.getState() = RUNNABLE

//myThread 출력
09:55:58.717 [     main] myThread = Thread[#21,myThread,5,main]
09:55:58.717 [     main] myThread.threadId() = 21
09:55:58.717 [     main] myThread.getName() = myThread
09:55:58.717 [     main] myThread.getPriority() = 5
09:55:58.717 [     main] myThread.getThreadGroup() =
java.lang.ThreadGroup[name=main,maxpri=10]
09:55:58.717 [     main] myThread.getState() = NEW
```

1. 스레드 생성

스레드를 생성할 때는 실행할 `Runnable` 인터페이스의 구현체와, 스레드의 이름을 전달할 수 있다.

```
Thread myThread = new Thread(new HelloRunnable(), "myThread");
```

- **Runnable 인터페이스**: 실행할 작업을 포함하는 인터페이스이다. `HelloRunnable` 클래스는 `Runnable` 인터페이스를 구현한 클래스이다.
- **스레드 이름**: `"myThread"` 라는 이름으로 스레드를 생성한다. 이 이름은 디버깅이나 로깅 목적으로 유용하다. 참고로 이름을 생략하면 `Thread-0`, `Thread-1` 과 같은 임의의 이름이 생성된다.

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.

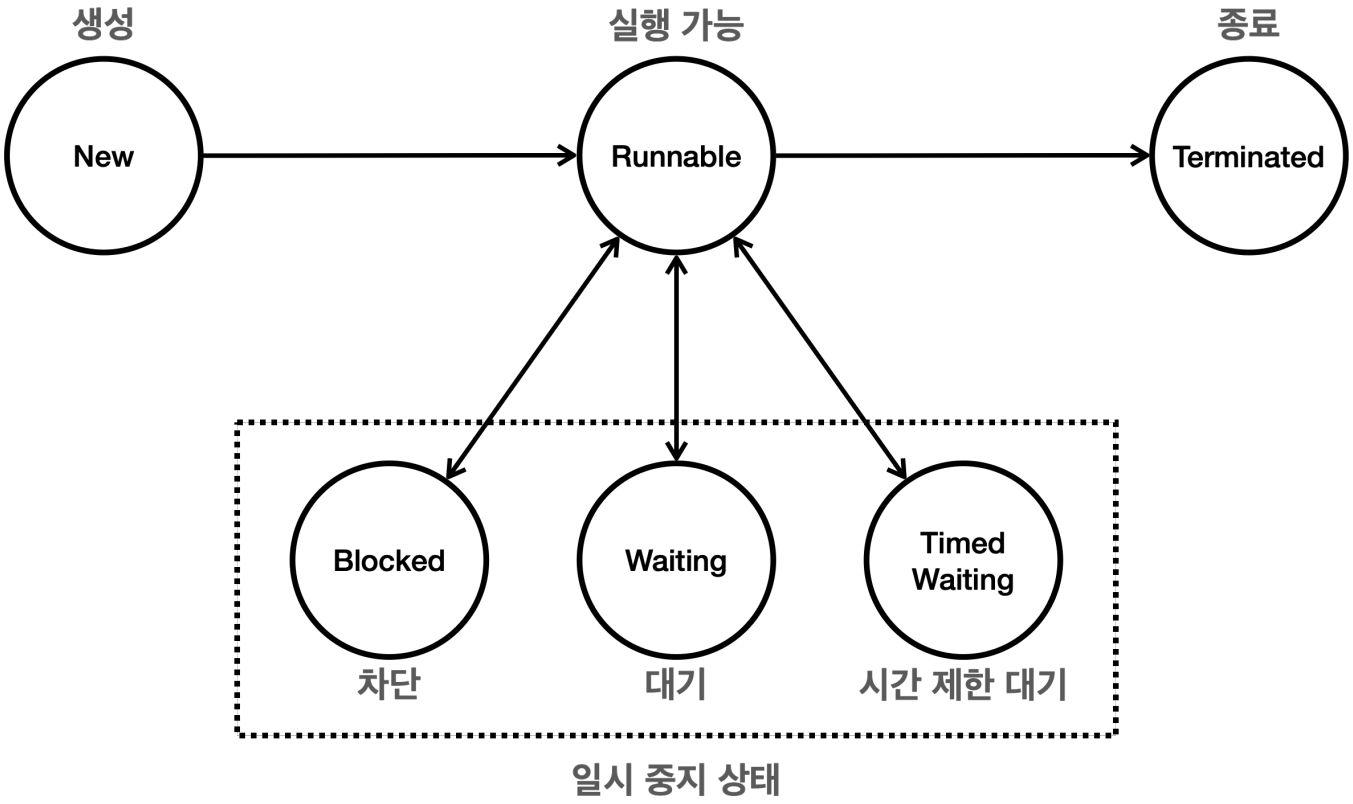
출력 결과를 보면 `main` 스레드는 실행 중이기 때문에 `RUNNABLE` 상태이다.
`myThread`는 생성하고 아직 시작하지 않았기 때문에, `NEW` 상태이다.

```
09:55:58.716 [     main] mainThread.getState() = RUNNABLE
...
09:55:58.717 [     main] myThread.getState() = NEW
```

스레드의 생명 주기 - 설명

스레드는 생성하고 시작하고, 종료되는 생명주기를 가진다. 스레드의 생명 주기에 대해 자세히 알아보자.

그림 - 스레드 생명 주기



스레드의 상태

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

> **참고**: 자바에서 스레드의 **일시 중지 상태들(Suspended States)**이라는 상태는 없다. 스레드가 기다리는 상태들을 묶어서 쉽게 설명하기 위해 사용한 용어이다.

자바 스레드(Thread)의 생명 주기는 여러 상태(state)로 나뉘어지며, 각 상태는 스레드가 실행되고 종료되기까지의 과정을 나타낸다.
자바 스레드의 생명 주기를 자세히 알아보자.

1. **New (새로운 상태)**
   - 스레드가 생성되고 아직 시작되지 않은 상태이다.
   - 이 상태에서는 `Thread` 객체가 생성되지만, `start()` 메서드가 호출되지 않은 상태이다.
   - 예: `Thread thread = new Thread(runnable);`

2. **Runnable (실행 가능 상태)**
    - 스레드가 실행될 준비가 된 상태이다. 이 상태에서 스레드는 실제로 CPU에서 실행될 수 있다.
    - `start()` 메서드가 호출되면 스레드는 이 상태로 들어간다.
    - 예: `thread.start();`
    - 이 상태는 스레드가 실행될 준비가 되어 있음을 나타내며, 실제로 CPU에서 실행될 수 있는 상태이다. 그러나 Runnable 상태에 있는 모든 스레드가 동시에 실행되는 것은 아니다. 운영체제의 스케줄러가 각 스레드에 CPU 시간을 할당하여 실행하기 때문에, Runnable 상태에 있는 스레드는 스케줄러의 실행 대기열에 포함되어 있다가 차례로 CPU에서 실행된다.
    - 참고로 운영체제 스케줄러의 실행 대기열에 있든, CPU에서 실제 실행되고 있든 모두 `RUNNABLE` 상태이다. 자바에서 둘을 구분해서 확인할 수는 없다.
    - 보통 실행 상태라고 부른다.

3. **Blocked (차단 상태)**
    - 스레드가 다른 스레드에 의해 동기화 락을 얻기 위해 기다리는 상태이다.
    - 예를 들어, `synchronized` 블록에 진입하기 위해 락을 얻어야 하는 경우 이 상태에 들어간다.
    - 예: `synchronized (lock) { ... }` 코드 블록에 진입하려고 할 때, 다른 스레드가 이미 `lock`의 락을 가지고 있는 경우.
    - 지금은 이런 상태가 있다 정도만 알아두자. 이 부분은 뒤에서 자세히 다룬다.

4. **Waiting (대기 상태)**
    - 스레드가 다른 스레드의 특정 작업이 완료되기를 무기한 기다리는 상태이다.
    - `wait()`, `join()` 메서드가 호출될 때 이 상태가 된다.
    - 스레드는 다른 스레드가 `notify()` 또는 `notifyAll()` 메서드를 호출하거나, `join()`이 완료될 때까지 기다린다.
    - 예: `object.wait();`
    - 지금은 이런 상태가 있다 정도만 알아두자. 이 부분은 뒤에서 자세히 다룬다.

5. **Timed Waiting (시간 제한 대기 상태)**
    - 스레드가 특정 시간 동안 다른 스레드의 작업이 완료되기를 기다리는 상태이다.
    - `sleep(long millis)`, `wait(long timeout)`, `join(long millis)` 메서드가 호출될 때 이 상태가 된다.
    - 주어진 시간이 경과하거나 다른 스레드가 해당 스레드를 깨우면 이 상태에서 벗어난다.
    - 예: `Thread.sleep(1000);`
    - 지금은 이런 상태가 있다 정도만 알아두자. 이 부분은 뒤에서 자세히 다룬다.

6. **Terminated (종료 상태)**

- 스레드의 실행이 완료된 상태이다.
- 스레드가 정상적으로 종료되거나, 예외가 발생하여 종료된 경우 이 상태로 들어간다.
- 스레드는 한 번 종료되면 다시 시작할 수 없다.

자바 스레드의 상태 전이 과정

1. **New → Runnable**: `start()` 메서드를 호출하면 스레드가 `Runnable` 상태로 전이된다.
2. **Runnable → Blocked/Waiting/Timed Waiting**: 스레드가 락을 얻지 못하거나, `wait()` 또는 `sleep()` 메서드를 호출할 때 해당 상태로 전이된다.
3. **Blocked/Waiting/Timed Waiting → Runnable**: 스레드가 락을 얻거나, 기다림이 완료되면 다시 `Runnable` 상태로 돌아간다.
4. **Runnable → Terminated**: 스레드의 `run()` 메서드가 완료되면 스레드는 `Terminated` 상태가 된다.

스레드의 생명 주기 - 코드

스레드 생명 주기를 코드로 확인해보자. 스레드가 생성, 실행, 대기 및 종료 상태로 변할 때마다 해당 상태를 로그로 출력한다. 이를 통해 스레드의 생명주기를 이해해보자.

```
package thread.control;

import static util.MyLogger.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()); // TIMED_WAITING
        Thread.sleep(4000);
        log("myThread.state5 = " + thread.getState()); // TERMINATED
        log("end");
    }

    static class MyRunnable implements Runnable {
        public void run() {
```

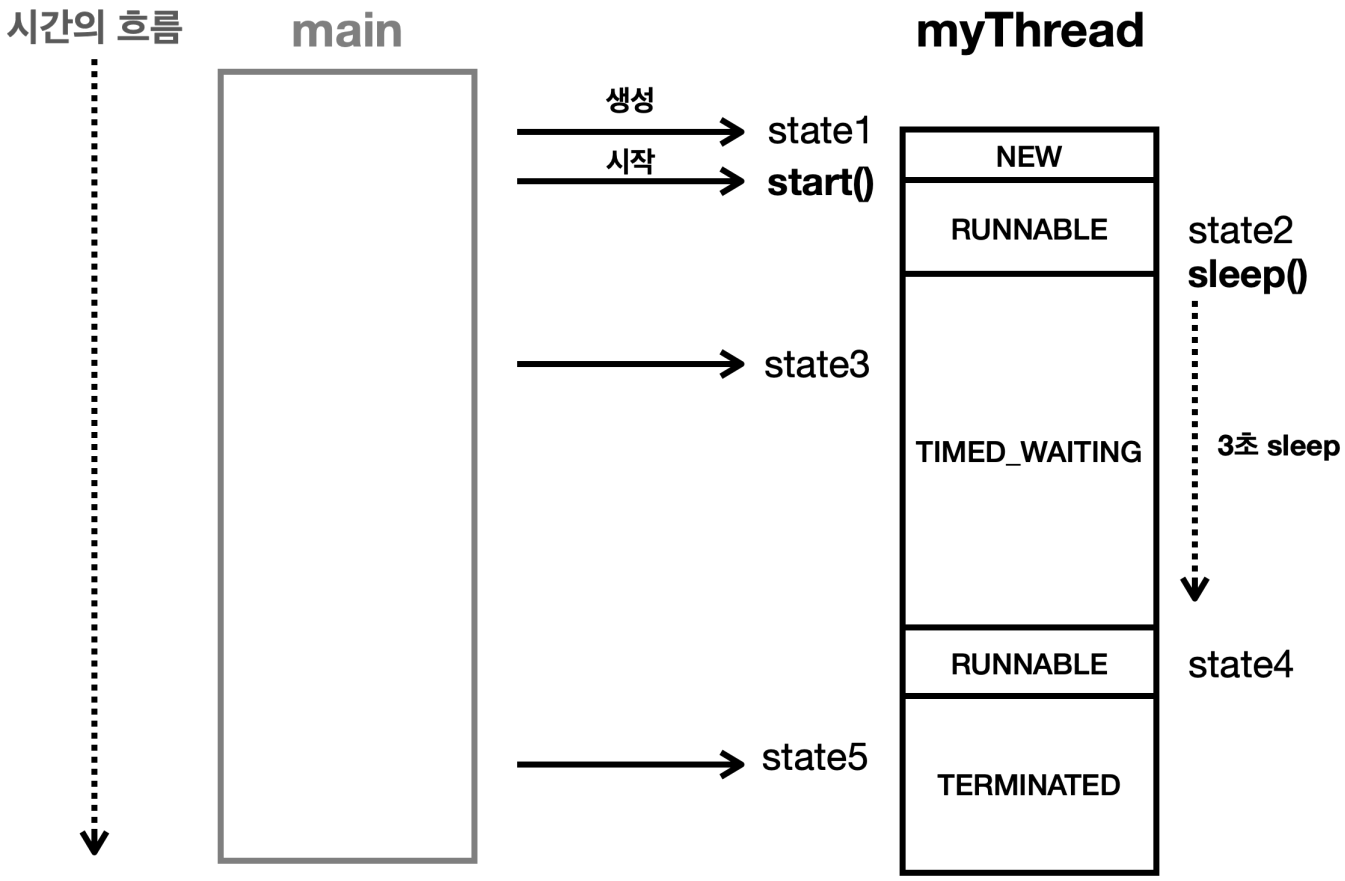
```
            try {
                log("start");
                log("myThread.state2 = " +
Thread.currentThread().getState()); // RUNNABLE
                log("sleep() start");
                Thread.sleep(3000);
                log("sleep() end");
                log("myThread.state4 = " +
Thread.currentThread().getState()); // RUNNABLE
                log("end");
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

- `Thread.currentThread()`를 호출하면 해당 코드를 실행하는 스레드 객체를 조회할 수 있다.
- `Thread.sleep()`: 해당 코드를 호출한 스레드는 `TIMED_WAITING` 상태가 되면서 특정 시간 만큼 대기한다. 시간은 밀리초(ms) 단위이다. 1밀리초 = 1/1000 초, 1000밀리초 = 1초이다.
- `Thread.sleep()`은 `InterruptedException`이라는 체크 예외를 던진다. 따라서 체크 예외를 잡아서 처리하거나 던져야 한다.
  - `run()` 메서드 안에서는 체크 예외를 반드시 잡아야 한다. 이 부분은 바로 뒤에서 설명한다.
- `InterruptedException`은 인터럽트가 걸릴 때 발생하는데, 인터럽트는 뒤에서 알아본다. 지금은 체크 예외가 발생한다 정도만 이해하면 충분하다.

실행 결과

```
11:40:31.503 [     main] myThread.state1 = NEW
11:40:31.505 [     main] myThread.start()
11:40:31.505 [ myThread] start
11:40:31.505 [ myThread] myThread.state2 = RUNNABLE
11:40:31.505 [ myThread] sleep() start
11:40:32.507 [     main] myThread.state3 = TIMED_WAITING
11:40:34.510 [ myThread] sleep() end
11:40:34.512 [ myThread] myThread.state4 = RUNNABLE
11:40:34.512 [ myThread] end
11:40:36.511 [     main] myThread.state5 = TERMINATED
11:40:36.512 [     main] end
```

실행 상태 그림

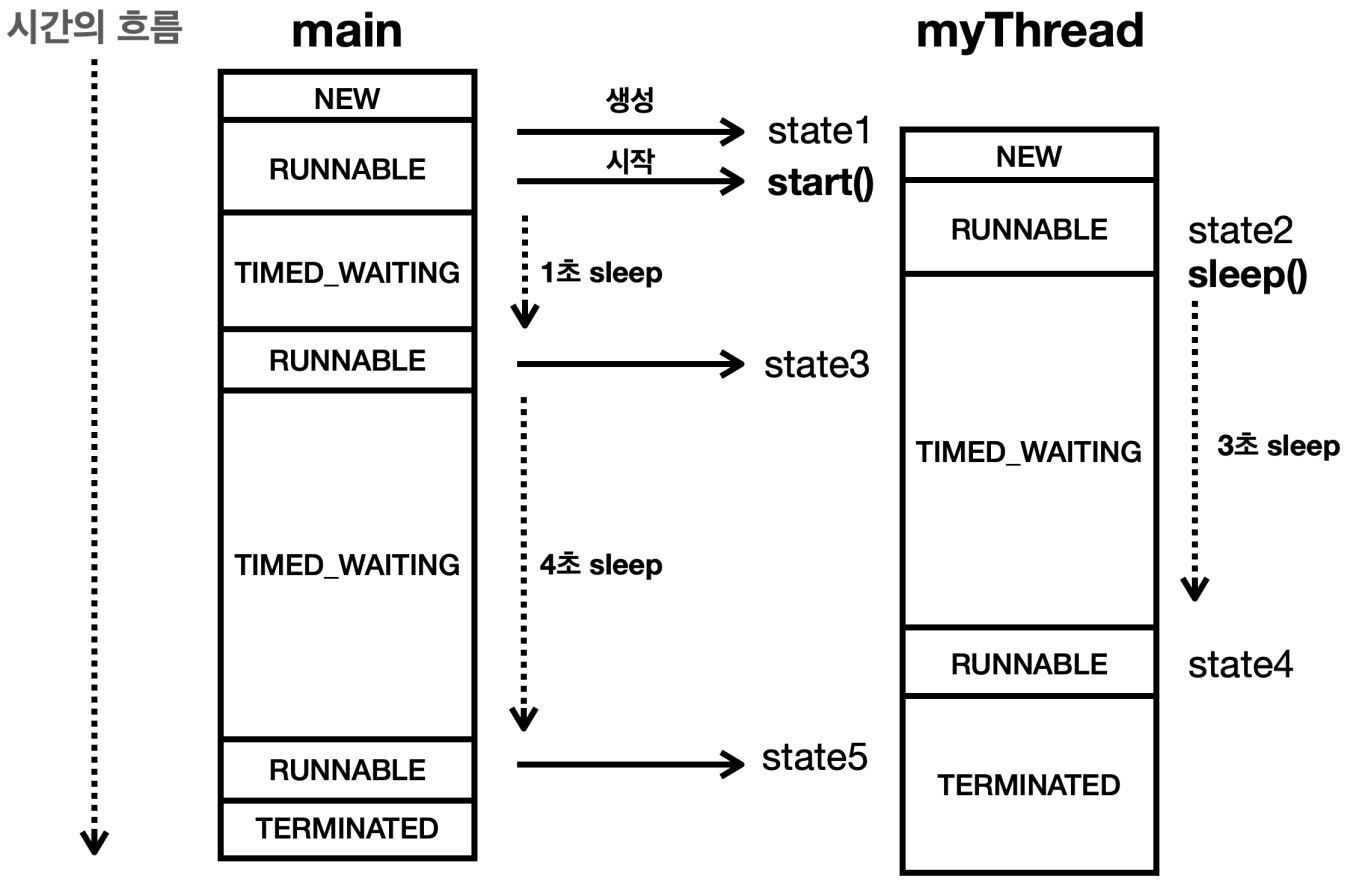


- `main` 스레드의 상태는 생략했다. 여기서는 `myThread` 스레드의 상태에 집중하자

- **state1 = NEW**
  - `main` 스레드를 통해 `myThread` 객체를 생성한다. 스레드 객체만 생성하고 아직 `start()`를 호출하지 않았기 때문에 `NEW` 상태이다.
- **state2 = RUNNABLE**
  - `myThread.start()`를 호출해서 `myThread`를 실행 상태로 만든다. 따라서 `RUNNABLE` 상태가 된다. 참고로 실행 상태가 너무 빨리 지나가기 때문에 `main` 스레드에서 `myThread`의 상태를 확인하기는 어렵다. 대신에 자기 자신인 `myThread`에서 실행 중인 자신의 상태를 확인했다.
- **state3 = TIMED_WAITING**
  - `Thread.sleep(3000)` : 해당 코드를 호출한 스레드는 3000ms (3초)간 대기한다. `myThread`가 해당 코드를 호출했으므로 3초간 대기하면서 `TIMED_WAITING` 상태로 변한다.
  - 참고로 이때 `main` 스레드가 `myThread`의 `TIMED_WAITING` 상태를 확인하기 위해 1초간 대기하고 상태를 확인했다.
- **state4 = RUNNABLE**
  - `myThread`는 3초의 시간 대기 후 `TIMED_WAITING` 상태에서 빠져나와 다시 실행될 수 있는 `RUNNABLE` 상태로 바뀐다.
- **state5 = TERMINATED**
  - `myThread`가 `run()` 메서드를 실행 종료하고 나면 `TERMINATED` 상태가 된다.
  - `myThread` 입장에서 `run()`이 스택에 남은 마지막 메서드인데, `run()`까지 실행되고 나면 스택이 완전

히 비워진다. 이렇게 스택이 비워지면 해당 스택을 사용하는 스레드도 종료된다.

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 이유를 알아보자.

`Runnable` 인터페이스는 다음과 같이 정의되어 있다.

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

자바에서 메서드를 재정의 할 때, 재정의 메서드가 지켜야할 예외와 관련된 규칙이 있다.

- **체크 예외**

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

`Runnable` 인터페이스의 `run()` 메서드는 아무런 체크 예외를 던지지 않는다. 따라서 `Runnable` 인터페이스의 `run()` 메서드를 재정의 하는 곳에서는 체크 예외를 밖으로 던질 수 없다. 다음 코드를 실행하면 컴파일 오류가 발생한다.

```
package thread.control;

public class CheckedExceptionMain {

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

    static class CheckedRunnable implements Runnable {

        @Override
        public void run() /*throws Exception*/ { // 주석 풀면 예외 발생
            //throw new Exception(); // 주석 풀면 예외 발생
        }
    }
}
```

- `main()`은 체크 예외를 밖으로 던질 수 있다.
- `run()`은 체크 예외를 밖으로 던질 수 없다.

예를 들어 다음 코드의 `InterruptedException`도 체크 예외이므로 던질 수 없다. 컴파일 오류가 발생한다.

```
static class MyRunnable implements Runnable {
    public void run() throws InterruptedException {
        Thread.sleep(3000);
    }
}
```

자바는 왜 이런 제약을 두는 것일까?
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, 예상하지 못한 런타임 오류를 초래할 수 있다.

다음 예를 보자. 실제 작동하는 코드는 아니다.

```
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 예외 재정의 규칙

- 자식 클래스에 재정의된 메서드는 부모 메서드가 던질 수 있는 체크 예외의 하위 타입만을 던질 수 있다.
- 원래 메서드가 체크 예외를 던지지 않는 경우, 재정의된 메서드도 체크 예외를 던질 수 없다.

안전한 예외 처리

체크 예외를 `run()` 메서드에서 던질 수 없도록 강제함으로써, 개발자는 반드시 체크 예외를 try-catch 블록 내에서 처리하게 된다. 이는 예외 발생 시 예외가 적절히 처리되지 않아서 프로그램이 비정상 종료되는 상황을 방지할 수 있다. 특히 멀티스레딩 환경에서는 예외 처리를 강제함으로써 스레드의 안정성과 일관성을 유지할 수 있다.
하지만 이전에 자바 예외 처리 강의에서 설명했듯이, 체크 예외를 강제하는 이런 부분들은 자바 초창기 기조이고, 최근에는 체크 예외보다는 언체크(런타임) 예외를 선호한다.

Sleep 유틸리티

강의에서 `Thread.sleep()`을 자주 사용할 예정인데, 이 코드는 `InterruptedException` 체크 예외를 발생시킨다.

학습용 예제의 `run()` 메서드 안에서 다음과 같이 `try ~ catch`를 계속 사용하는 것은 상당히 번거롭다.

호출 코드 예시 - 기존

```
void run() {
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

다음과 같이 체크 예외를 런타임 예외로 변경하는 간단한 유틸리티를 만들어 사용하자.

ThreadUtils.sleep()

```
package util;

import static util.MyLogger.log;

public abstract class ThreadUtils {

    public static void sleep(long millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            log("인터럽트 발생, " + e.getMessage());
            throw new RuntimeException(e);
        }
    }
}
```

호출 코드 예시 - 유틸리티 사용

```
import static util.ThreadUtils.sleep;

void run() {
    sleep(3000);
}
```

join - 시작

앞서 `Thread.sleep()` 메서드를 통해 `TIMED_WAITING` 상태를 알아보았다. 이번에는 `join()` 메서드를 통해 `WAITING`(대기 상태)가 무엇이고 왜 필요한지 알아보자.

Waiting (대기 상태)

- 스레드가 다른 스레드의 특정 작업이 완료되기를 무기한 기다리는 상태이다.

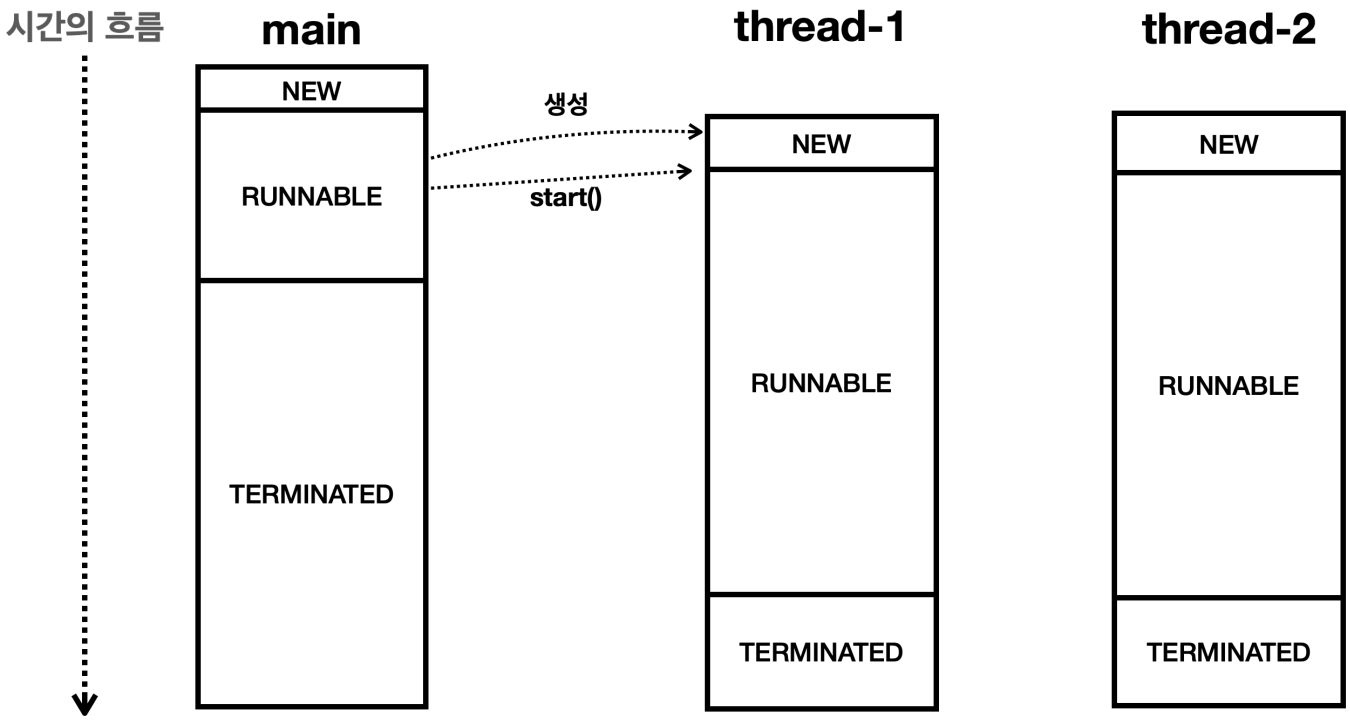
먼저 스레드로 특정 작업을 수행하는 간단한 예제를 하나 만들어보자.

```
package thread.control.join;

import static util.MyLogger.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");
        }
    }
}
```

실행 결과

```
15:13:40.734 [     main] Start
15:13:40.736 [     main] End
15:13:40.736 [ thread-2] 작업 시작
15:13:40.736 [ thread-1] 작업 시작
15:13:42.741 [ thread-1] 작업 완료
15:13:42.741 [ thread-2] 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.

`thread-1`, `thread-2`는 각각 특정 작업을 수행한다. 작업 수행에 약 2초 정도가 걸린다고 가정하기 위해 `sleep()`을 사용해서 2초간 대기한다. (그림에서는 `RUNNABLE`로 표현했지만, 실제로는 `TIMED_WAITING` 상태이다.)

`sleep()` 메서드는 `Thread.sleep()` 대신에 앞서 만든 유틸리티를 `import static`으로 사용한다.

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

실행 결과를 보면 `main` 스레드가 먼저 종료되고, 그 다음에 `thread-1`, `thread-2`가 종료된다.

```
thread1.start();
thread2.start();
log("End");
```

`main` 스레드는 `thread-1`, `thread-2`를 실행하고 바로 자신의 다음 코드를 실행한다. 여기서 핵심은 `main` 스레드가 `thread-1`, `thread-2`가 끝날때까지 기다리지 않는다는 점이다. `main` 스레드는 단지 `start()`를 호출해서 다른 스레드를 실행만 하고 바로 자신의 다음 코드를 실행한다.

그런데 만약 `thread-1`, `thread-2`가 종료된 다음에 `main` 스레드를 가장 마지막에 종료하려면 어떻게 해야할까? 예를 들어서 `main` 스레드가 `thread-1`, `thread-2`에 각각 어떤 작업을 지시하고, 그 결과를 받아서 처리하고 싶다면 어떻게 해야할까?
이런 상황이 왜 필요한지 이해를 돕기위해 간단한 예제를 만들어보자.

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)

이제 코드를 작성해보자.

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

```java
import static util.MyLogger.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;
        log("task1 + task2 = " + sumAll);
        log("End");
    }

    static class SumTask implements Runnable {
        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=" + result);
```

```
        }
    }

}
```

`SumTask`는 계산의 시작값(`startValue`)과 계산의 마지막 값(`endValue`)을 가진다. 그리고 계산이 끝나면 그 결과를 `result` 필드에 담아둔다.
`main` 스레드는 `thread-1`, `thread-2`를 만들고 다음과 같이 작업을 할당한다.

- `thread-1`: `task1` - `1 ~ 50`까지 더하기
- `thread-2`: `taks2` - `51 ~ 100`까지 더하기

`thread-1`은 `task1` 인스턴스의 `run()`을 실행하고, `thread-2`는 `task2` 인스턴스의 `run()`을 실행한다. 각각의 스레드는 계산 결과를 `result` 멤버 변수에 보관한다.

`run()`에서 수행하는 계산이 2초 정도는 걸리는 복잡한 계산이라고 가정하자. 그래서 `sleep(2000)`으로 설정했다. 여기서는 약 2초 후에 계산이 완료되고 `result`에 결과가 담긴다.

`main` 스레드는 `thread1`, `thread2`에 작업을 지시한 다음에 작업의 결과인 `task1.result`, `task2.result`를 얻어서 사용한다.

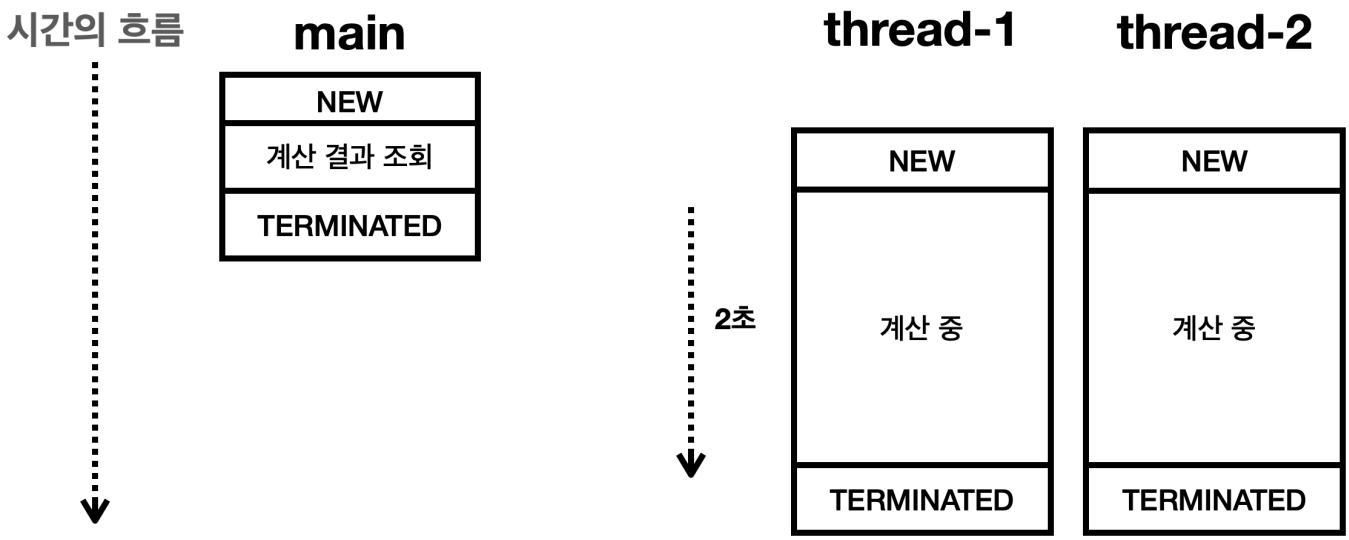
```
thread1.start();
thread2.start();

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

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

실행 결과

```
15:36:28.347 [     main] Start
15:36:28.349 [ thread-1] 작업 시작
15:36:28.349 [ thread-2] 작업 시작
15:36:28.352 [     main] task1.result = 0
15:36:28.352 [     main] task2.result = 0
15:36:28.352 [     main] task1 + task2 = 0
15:36:28.352 [     main] End
15:36:30.355 [ thread-1] 작업 완료 result=1275
15:36:30.355 [ thread-2] 작업 완료 result=3775
```

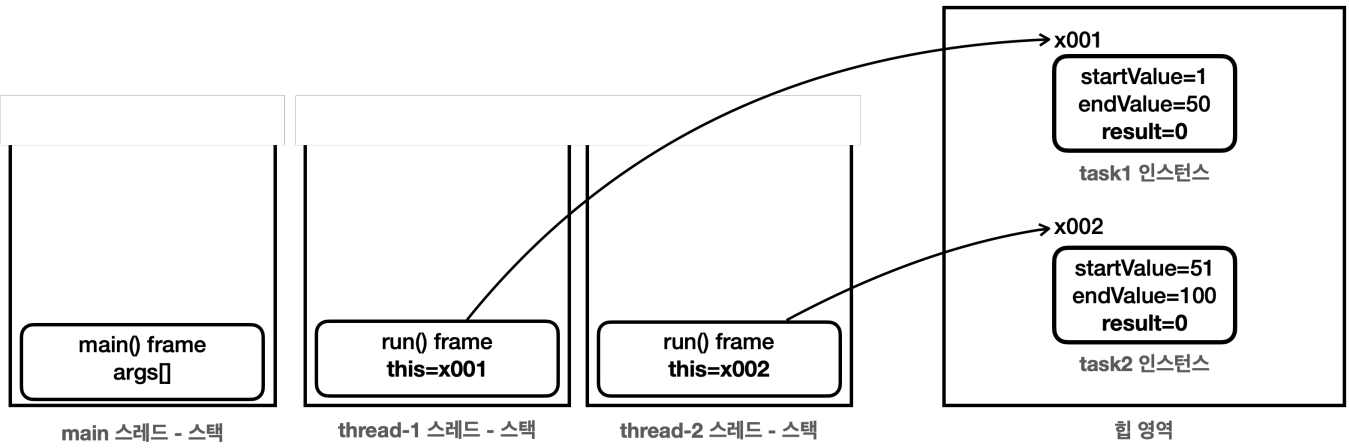
그런데 실행 결과를 보면 기대와 다르게 `task1.result`, `task2.result` 모두 0으로 나온다. 그리고 `task1 + task2`의 결과도 `0`으로 나온다. 계산이 전혀 진행되지 않았다. 이 부분을 자세히 분석해보자.

실행 결과 분석

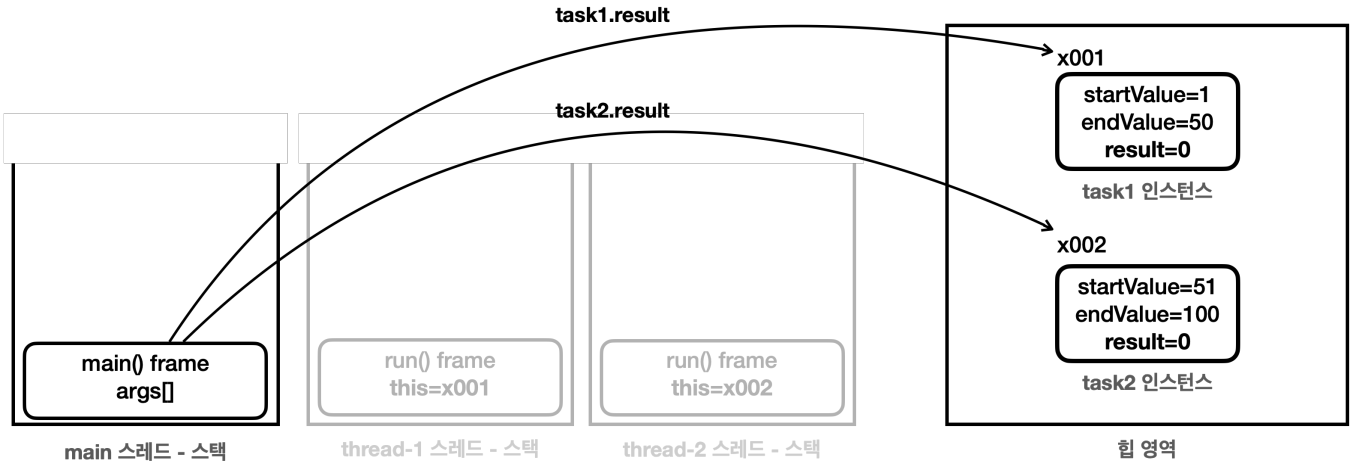


`main` 스레드는 `thread-1`, `thread2`에 작업을 지시하고, `thread-1`, `thread2`가 계산을 완료하기도 전에 먼저 계산 결과를 조회했다. 참고로 `thread-1`, `thread-2`가 계산을 완료하는데는 2초 정도의 시간이 걸린다. 따라서 결과가 `task1 + task2 = 0`으로 출력된다.

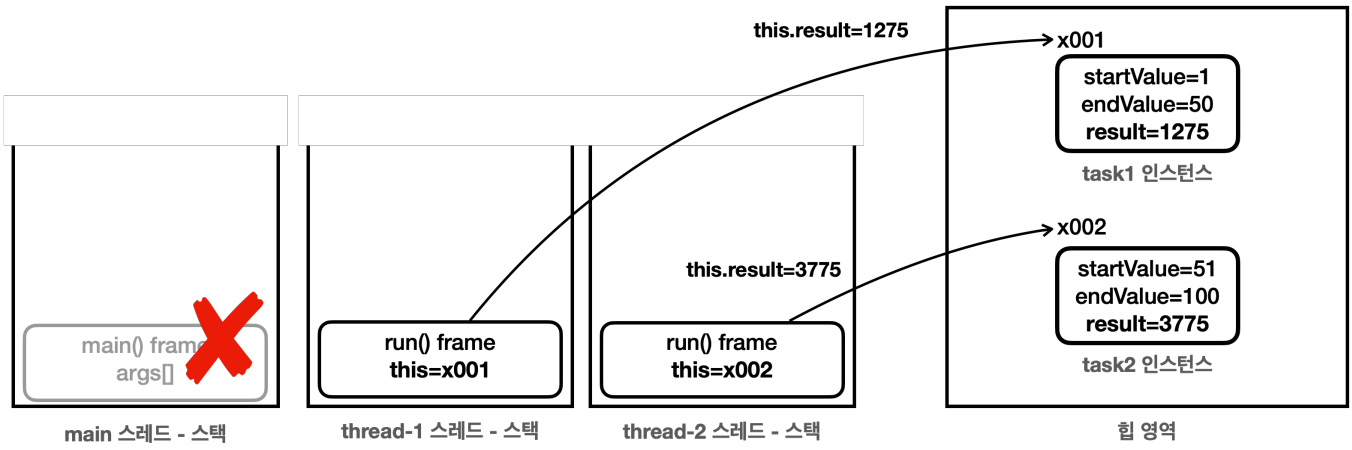
이 부분을 메모리 구조로 좀 더 자세히 살펴보자.



- 프로그램이 처음 시작되면 `main` 스레드는 `thread-1`, `thread-2`를 생성하고 `start()`로 실행한다.
- `thread-1`, `thread-2`는 각각 자신에게 전달된 `SumTask` 인스턴스의 `run()` 메서드를 스택에 올리고 실행한다.
  - `thread-1`은 `x001` 인스턴스의 `run()` 메서드를 실행한다.
  - `thread-2`는 `x002` 인스턴스의 `run()` 메서드를 실행한다.



- `main` 스레드는 두 스레드를 시작한 다음에 바로 `task1.result`, `task2.result`를 통해 인스턴스에 있는 결과 값을 조회한다. 참고로 `main` 스레드가 실행한 `start()` 메서드는 스레드의 실행이 끝날 때 까지 기다리지 않는다! 다른 스레드를 실행만 해두고, 자신의 다음 코드를 실행할 뿐이다!
- `thread-1`, `thread-2`가 계산을 완료해서, `result`에 연산 결과를 담을 때 까지는 약 2초 정도의 시간이 걸린다. `main` 스레드는 계산이 끝나기 전에 `result`의 결과를 조회한 것이다. 따라서 `0` 값이 출력된다.



- 2초가 지난 이후에 `thread-1`, `thread-2`는 계산을 완료한다.
- 이때 `main` 스레드는 이미 자신의 코드를 모두 실행하고 종료된 상태이다.
- `task1` 인스턴스의 `result`에는 `1275`가 담겨있고, `task2` 인스턴스의 `result`에는 `3775`가 담겨있다.

여기서 문제의 핵심은 `main` 스레드가 `thread-1`, `thread-2`의 계산이 끝날 때 까지 기다려야 한다는 점이다. 그럼 어떻게 해야 `main` 스레드가 기다릴 수 있을까?

참고 - this의 비밀

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 참조값을

스택 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`를 참고해서 필드에 접근한다.

정리하면 `this`는 호출된 인스턴스 메서드가 소속된 객체를 가리키는 참조이며, 이것이 스택 프레임 내부에 저장되어 있다.

join - sleep 사용

특정 스레드를 기다리게 하는 가장 간단한 방법은 `sleep()`을 사용하는 것이다.

```
package thread.control.join;

import static util.MyLogger.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();

        // 정확한 타이밍을 맞추어 기다리기 어려움
        log("main 스레드 sleep()");
        sleep(3000);
        log("main 스레드 깨어남");

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

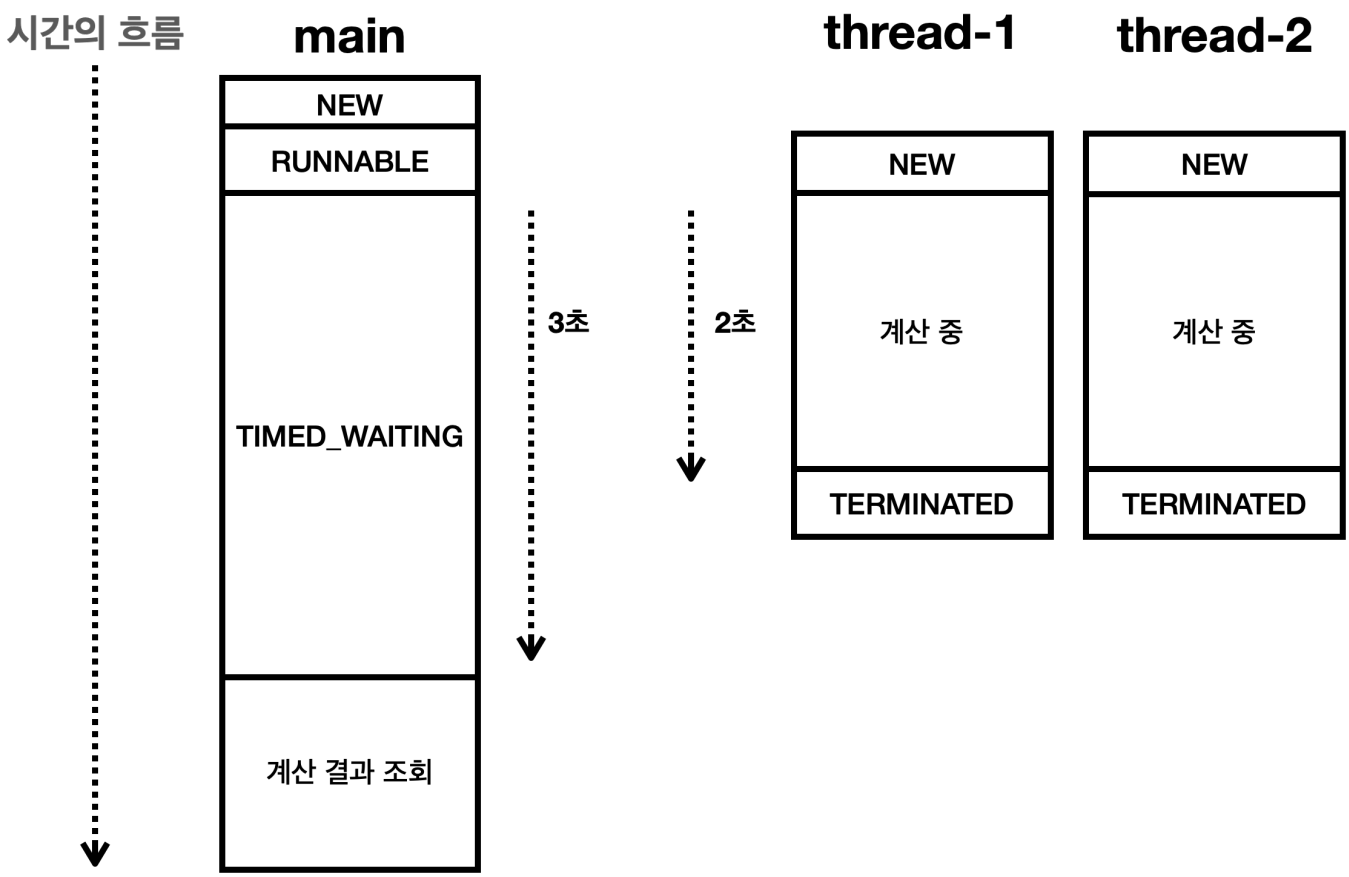
```
        int sumAll = task1.result + task2.result;
        log("task1 + task2 = " + sumAll);
        log("End");
    }

    static class SumTask implements Runnable {
        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 = " + result);
        }
    }

}
```

- `main` 스레드가 `sleep(3000)`을 사용해서 3초간 대기한다.

실행 결과

```
16:28:05.002 [     main] Start
16:28:05.004 [     main] main 스레드 sleep()
16:28:05.004 [ thread-1] 작업 시작
16:28:05.004 [ thread-2] 작업 시작
16:28:07.015 [ thread-1] 작업 완료 result = 1275
16:28:07.015 [ thread-2] 작업 완료 result = 3775
16:28:08.005 [     main] main 스레드 깨어남
16:28:08.007 [     main] task1.result = 1275
16:28:08.008 [     main] task2.result = 3775
```

```
16:28:08.008 [     main] task1 + task2 = 5050
16:28:08.008 [     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 사용해서 무작정? 기다리는 방법은 대기 시간에 손해도 보고, 또 `thread-1`, `thread-2`의 수행시간이 달라지는 경우에는 정확한 타이밍을 맞추기 어렵다.

더 나은 방법은 `thread-1`, `thread-2`가 계산을 끝내고 종료될 때 까지 `main` 스레드가 기다리는 방법이다.
예를 들어서 `main` 스레드가 반복문을 사용해서 `thread-1`, `thread-2`의 상태가 `TERMINATED`가 될 때 까지 계속 확인하는 방법이 있다.

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

이런 방법은 번거롭고 또 계속되는 반복문은 CPU 연산을 사용한다. 이때 `join()` 메서드를 사용하면 깔끔하게 문제를 해결할 수 있다.

join - join 사용

```
package thread.control.join;

import static util.MyLogger.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();

        // 스레드가 종료될 때 까지 대기
        log("join() - main 스레드가 thread1, thread2 종료까지 대기");
        thread1.join();
        thread2.join();
        log("main 스레드 대기 완료");

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

        int sumAll = task1.result + task2.result;
        log("task1 + task2 = " + sumAll);
        log("End");
    }

    static class SumTask implements Runnable {
        int startValue;
        int endValue;
        int result = 0;

        public SumTask(int startValue, int endValue) {
```

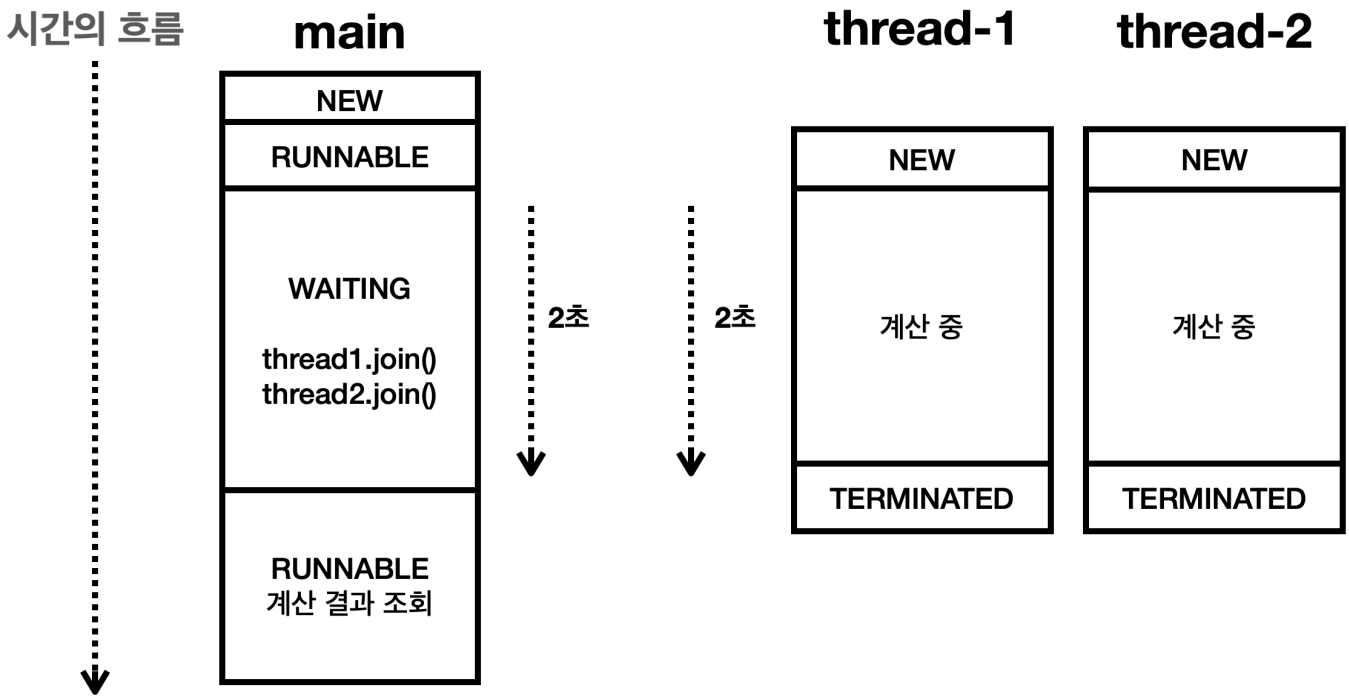
```
            this.startValue = startValue;
            this.endValue = endValue;
        }

        @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 = " + result);
        }
    }

}
```

- `join()`은 `InterruptedException`을 던진다. `InterruptedException`에 대해서는 뒤에서 설명한다.

실행 결과

```
16:46:54.788 [     main] Start
16:46:54.790 [ thread-1] 작업 시작
16:46:54.790 [ thread-2] 작업 시작
16:46:54.790 [     main] join() - main 스레드가 thread1, thread2 종료까지 대기
16:46:56.801 [ thread-2] 작업 완료 result = 3775
16:46:56.801 [ thread-1] 작업 완료 result = 1275
16:46:56.802 [     main] main 스레드 대기 완료
16:46:56.803 [     main] task1.result = 1275
16:46:56.803 [     main] task2.result = 3775
16:46:56.804 [     main] task1 + task2 = 5050
16:46:56.804 [     main] End
```

- 실행 결과를 보면 정확하게 `5050`이 계산된 것을 확인할 수 있다.



`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();
```

예를 들어서 `thread-1`이 아직 종료되지 않았다면 `main` 스레드는 `thread1.join()` 코드 안에서 더는 진행하지 않고 멈추어 기다린다. 이후에 `thread-1`이 종료되면 `main` 스레드는 `RUNNABLE` 상태가 되고 다음 코드로 이동한다.
이때 `thread-2`이 아직 종료되지 않았다면 `main` 스레드는 `thread2.join()` 코드 안에서 진행하지 않고 멈추어 기다린다. 이후에 `thread-2`이 종료되면 `main` 스레드는 `RUNNABLE` 상태가 되고 다음 코드로 이동한다.
이 경우 `thread-1`이 종료되는 시점에 `thread-2`도 거의 같이 종료되기 때문에 `thread2.join()`은 대기하지 않고 바로 빠져나온다.

Waiting (대기 상태)

- 스레드가 다른 스레드의 특정 작업이 완료되기를 무기한 기다리는 상태이다.
- `join()`을 호출하는 스레드는 대상 스레드가 `TERMINATED` 상태가 될 때 까지 대기한다. 대상 스레드가 `TERMINATED` 상태가 되면 호출 스레드는 다시 `RUNNABLE` 상태가 되면서 다음 코드를 수행한다.

이렇듯 특정 스레드가 완료될 때 까지 기다려야 하는 상황이라면 `join()`을 사용하면 된다.

하지만 `join()`의 단점은 다른 스레드가 완료될 때 까지 무기한 기다리는 단점이 있다. 비유를 하자면 맛집에 한 번 줄을 서면 중간에 포기하지 못하고 자리가 날 때 까지 무기한 기다려야 한다. 만약 다른 스레드의 작업을 일정 시간 동안만

기다리고 싶다면 어떻게 해야할까?

join - 특정 시간 만큼만 대기

`join()`은 두 가지 메서드가 있다.

- `join()` : 호출 스레드는 대상 스레드가 완료될 때 까지 무한정 대기한다.
- `join(ms)` : 호출 스레드는 특정 시간 만큼만 대기한다. 호출 스레드는 지정한 시간이 지나면 다시 `RUNNABLE` 상태가 되면서 다음 코드를 수행한다.

예제로 알아보자. 예제를 단순화 하기 위해 스레드는 1개만 만들고, 작업도 하나만 실행하자.

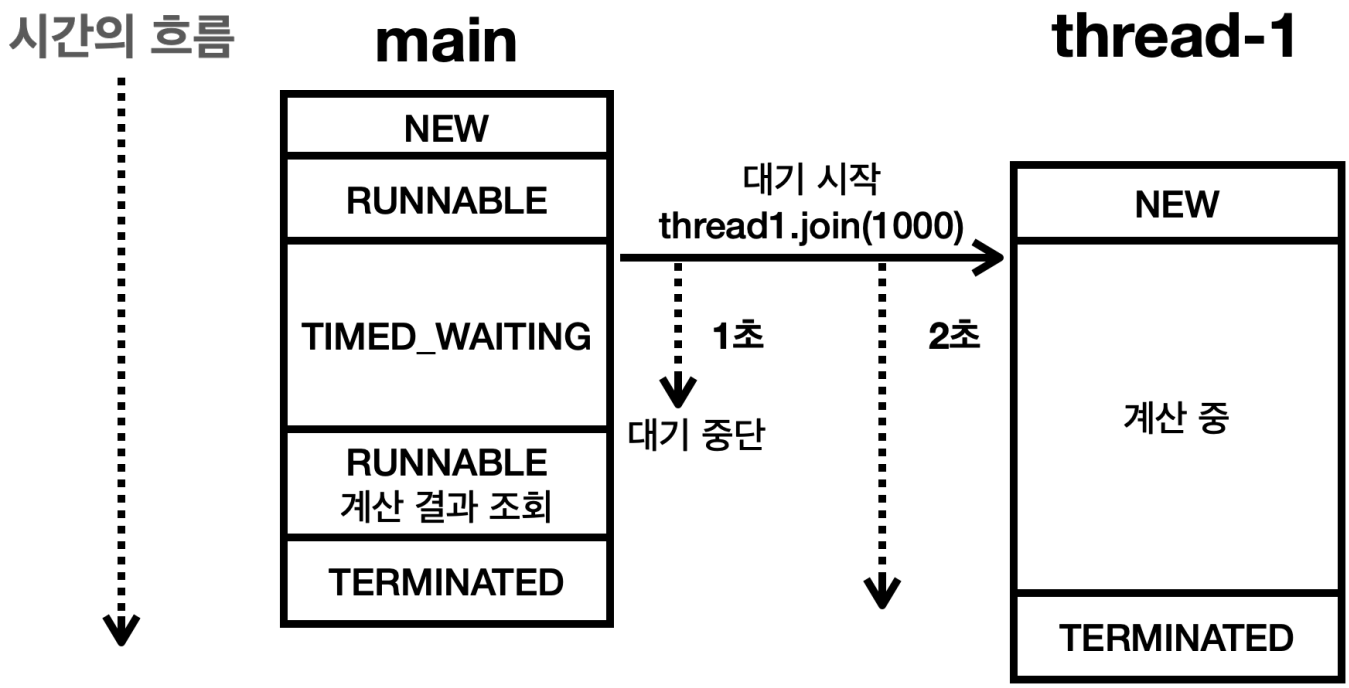
```
package thread.control.join;

import static util.MyLogger.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();

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

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

    static class SumTask implements Runnable {
        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 = " + result);
        }
    }

}
```

- 별도의 스레드에서 `1 ~ 50`까지 더하고, 그 결과를 조회한다.
- `join(1000)`을 사용해서 1초만 대기한다.

실행 결과

```
17:34:54.572 [     main] Start
17:34:54.575 [     main] join(1000) - main 스레드가 thread1 종료까지 1초 대기
17:34:54.575 [ thread-1] 작업 시작
17:34:55.580 [     main] main 스레드 대기 완료
17:34:55.585 [     main] task1.result = 0
17:34:56.580 [ thread-1] 작업 완료 result = 1275
```



- `main` 스레드는 `join(1000)`을 사용해서 `thread-1`을 1초간 기다린다.
  - 이때 `main` 스레드의 상태는 `WAITING`이 아니라 `TIMED_WAITING`이 된다.
  - 보통 무기한 대기하면 `WAITING` 상태가 되고, 특정 시간 만큼만 대기하는 경우 `TIMED_WAITING` 상태가 된다.
- `thread-1`의 작업에는 2초가 걸린다.
- 1초가 지나도 `thread-1`의 작업이 완료되지 않으므로, `main` 스레드는 대기를 중단한다. 그리고 `main` 스레드는 다시 `RUNNABLE` 상태로 바뀌면서 다음 코드를 수행한다.
  - 이때 `thread-1`의 작업이 아직 완료되지 않았기 때문에 `task1.result = 0`이 출력된다.
- `main` 스레드가 종료된 이후에 `thread-1`이 계산을 끝낸다. 따라서 `작업 완료 result = 1275`이 출력된다.

정리

다른 스레드가 끝날 때 까지 무한정 기다려야 한다면 `join()`을 사용하고, 다른 스레드의 작업을 무한정 기다릴 수 없다면 `join(ms)`를 사용하면 된다. 물론 기다리다 중간에 나오는 상황인데, 결과가 없다면 추가적인 오류 처리가 필요할 수 있다.

문제와 풀이

문제1 - join() 활용1

다음 코드를 작성하고, 코드를 실행하기 전에 **로그가 어떻게 출력될지 예측**해 보자. 그리고 **총 실행 시간이 얼마가 걸릴**

지 예측해 보자.

```
package thread.control.test;

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

public class JoinTest1Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new MyTask(), "t1");
        Thread t2 = new Thread(new MyTask(), "t2");
        Thread t3 = new Thread(new MyTask(), "t3");

        t1.start();
        t1.join();

        t2.start();
        t2.join();

        t3.start();
        t3.join();
        System.out.println("모든 스레드 실행 완료");
    }

    static class MyTask implements Runnable {

        @Override
        public void run() {
            for (int i = 1; i <= 3; i++) {
                log(i);
                sleep(1000);
            }
        }
    }
}
```

정답

```
10:26:59.939 [       t1] 1
10:27:00.946 [       t1] 2
10:27:01.951 [       t1] 3
10:27:02.955 [       t2] 1
```

```
10:27:03.961 [       t2] 2
10:27:04.967 [       t2] 3
10:27:05.973 [       t3] 1
10:27:06.978 [       t3] 2
10:27:07.984 [       t3] 3
모든 스레드 실행 완료
```

실행 시간: 9초

문제2 - join() 활용2

문제1의 코드를 변경해서 전체 실행 시간을 3초로 앞당겨보자. 실행 결과를 참고하자.

```
package thread.control.test;

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

public class JoinTest2Main {
    // 여기에 코드 작성
}
```

실행 결과

```
10:29:46.321 [       t1] 1
10:29:46.321 [       t3] 1
10:29:46.321 [       t2] 1
10:29:47.325 [       t2] 2
10:29:47.329 [       t3] 2
10:29:47.329 [       t1] 2
10:29:48.330 [       t3] 3
10:29:48.330 [       t1] 3
10:29:48.330 [       t2] 3
모든 스레드 실행 완료
```

실행 시간: 3초

정답

```
package thread.control.test;
```

```java
import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class JoinTest2Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new MyTask(), "t1");
        Thread t2 = new Thread(new MyTask(), "t2");
        Thread t3 = new Thread(new MyTask(), "t3");

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();
        System.out.println("모든 스레드 실행 완료");
    }

    static class MyTask implements Runnable {

        @Override
        public void run() {
            for (int i = 1; i <= 3; i++) {
                log(i);
                sleep(1000);
            }
        }
    }
}
```