15. 멀티스레딩과 동기화

Thread 생성 방법 (`extends Thread`, `implements Runnable`)

Java에서 멀티스레딩을 구현하려면 `Thread` 클래스를 **상속**하거나 `Runnable` 인터페이스를 **구현**하는 두 가지 주요 방법이 있다. 이 두 방식은 구조와 목적이 조금씩 다르며, 실무에서는 대부분 `Runnable`이나 `ExecutorService`를 더 많이 쓴다.

🟦 1. `extends Thread` 방식

✅ 개념

- `Thread` 클래스를 **상속**하여 `run()` 메서드를 오버라이드한다.
- 객체 생성 후 `start()`를 호출하면 `run()`이 비동기 실행됨.

✅ 코드 예시

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread 실행: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();  // run()이 아닌 start() 호출해야 별도 스레드에서 실행됨
    }
}
```

⚠️ 주의

- `run()`을 직접 호출하면 멀티스레딩 아님 (main 스레드에서 실행됨)
- Java는 단일 상속이므로 **이미 다른 클래스를 상속 중이라면 사용 불가**

🟩 2. `implements Runnable` 방식

✅ 개념

- `Runnable` 인터페이스는 `run()` 메서드만을 포함하는 함수형 인터페이스
- 일반 클래스에 `Runnable`을 구현시킨 후 `Thread`에 전달하여 실행

✅ 코드 예시

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable 실행: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

🟧 3. 두 방식 비교

항목	`extends Thread`	`implements Runnable`
상속	이미 다른 클래스 상속 중이면 불가	인터페이스이므로 다중 구현 가능
코드 구조	Thread 기능과 작업 내용이 한 클래스에 결합	작업 분리 가능 (Thread와 Runnable 분리)
실무 사용	거의 사용되지 않음	일반적으로 선호됨
재사용성	낮음	높음

🟥 4. 람다식과 함께 사용 (Runnable)

```
Thread t = new Thread(() -> {
    System.out.println("람다 기반 Runnable 스레드");
});
t.start();
```

🧩 추가: `Thread` vs `ExecutorService`

- `Thread`는 직접 스레드를 생성하고 관리
- `ExecutorService`는 **스레드 풀**을 통해 재사용 및 관리 → 리소스 절약

```
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(() -> System.out.println("풀에서 실행"));
```

✅ 요약

방식	장점	단점	추천 상황
`extends Thread`	간단한 테스트, 구조 명확	단일 상속 제한, 구조 결합	아주 간단한 테스트용 코드 등
`implements Runnable`	재사용성, 유연성, 병렬 설계 가능	코드가 길어질 수 있음	실무에서 대부분 사용
람다식	코드 간결, Java 8+에서 가능	복잡한 로직에는 부적합	짧은 작업, 이벤트 처리 등

Thread 상태와 생명주기

🟦 Thread의 생명주기 (Life Cycle)

Java 스레드는 다음 6가지 상태를 가질 수 있다:

```
NEW → RUNNABLE → RUNNING → BLOCKED / WAITING / TIMED_WAITING → TERMINATED
```

> 실질적으로는 JVM은 `RUNNABLE` 상태 하나로 운영하지만, OS 스케줄러의 관점에서 RUNNABLE ↔ RUNNING을 구분해서 설명하는 경우도 있다.

🟧 1. `NEW`

- `Thread` 객체가 생성되었지만 아직 `start()`가 호출되지 않은 상태

```
Thread t = new Thread(() -> {});
```

🟩 2. `RUNNABLE`

- `start()`를 호출한 후 실행 가능 상태가 됨
- **실행 중이 아닐 수도 있음** → CPU 스케줄링 대기 중

```
t.start();  // 실행 대기 상태
```

🟥 3. `RUNNING`

- OS 스케줄러가 CPU를 할당하여 `run()` 메서드를 실제로 실행 중인 상태
- Java에서는 명시적으로 이 상태를 감지하지 않음

🟦 4. `BLOCKED`

- **다른 스레드가 소유 중인 락(lock)을 기다리는 상태**
- `synchronized` 블록에 진입하려는데, 락이 점유된 경우

```
synchronized(obj) {
  // 다른 스레드가 obj를 점유 중이면 BLOCKED
}
```

🟧 5. `WAITING`

- 명시적으로 다른 스레드의 **신호 (`notify()` 등)** 를 기다리는 상태
- 무한정 대기

```
synchronized(lock) {
  lock.wait();  // WAITING 상태로 진입
}
```

- 반드시 `notify()` 또는 `notifyAll()`로 다시 깨워야 함

🟨 6. `TIMED_WAITING`

- 일정 시간 동안 기다리는 상태
- 시간 경과 후 자동으로 `RUNNABLE` 상태로 전환됨

메서드	의미
`Thread.sleep(ms)`	지정된 시간 동안 대기
`join(timeout)`	특정 스레드 종료를 기다림
`wait(timeout)`	`Object.wait()`의 시간 제한 버전

🟥 7. `TERMINATED` (`DEAD`)

- `run()` 메서드가 정상 종료되었거나 예외로 인해 비정상 종료됨

```
if (!thread.isAlive()) {
    System.out.println("스레드 종료됨");
}
```

🧭 전체 상태 전이도 (요약)

```
NEW
 ↓ start()
RUNNABLE
 ↓ (스케줄링됨)
RUNNING
 ↓
(종료) → TERMINATED
 ↓
(동기화 실패) → BLOCKED
 ↓
(wait 호출) → WAITING
 ↓
(sleep/join) → TIMED_WAITING
```

✅ 스레드 상태 확인

```
Thread.State state = t.getState();
System.out.println(state);  // 예: RUNNABLE, WAITING, etc.
```

💡 실전 팁

- `BLOCKED`, `WAITING`, `TIMED_WAITING` 상태는 **CPU를 거의 사용하지 않음**
- `RUNNABLE` 상태여도 **실행 중이 아닐 수 있음**
- 데드락/무한 대기 디버깅 시 `getState()`로 상태 점검
- `Thread.sleep()`은 `InterruptedException`을 반드시 처리해야 함

`synchronized` 키워드

🟦 1. `synchronized`란?

- 한 번에 **오직 하나의 스레드**만 임계 영역(critical section)에 접근하도록 보장
- 내부적으로 **monitor(모니터락, intrinsic lock)** 을 사용
- 객체 단위(lock object), 클래스 단위(lock class)로 동기화 가능

🟧 2. 사용 방식

✅ 인스턴스 메서드 동기화

```
public synchronized void increment() {
    count++;  // 여러 스레드에서 동시에 접근하면 데이터 손실 가능
}
```

- `this` 객체를 락으로 사용
- 같은 객체의 다른 synchronized 메서드도 동시에 실행 안 됨

✅ 블록 동기화 (`synchronized(obj)`)

```
public void increment() {
    synchronized(this) {
        count++;
    }
}
```

- 특정 객체(`this`, `lock`, etc.)를 명시적으로 락으로 사용
- 더 **세밀한 제어**가 가능

✅ static 메서드 동기화

```
public static synchronized void log(String msg) {
    // 클래스 전체를 락으로 잡음
}
```

- 클래스 수준의 락 (`Class.class`)을 사용
- 모든 인스턴스가 공유

```
synchronized(MyClass.class) {
    // 정적 자원 접근 보호
}
```

🟥 3. 예제: 동기화 없이 발생하는 문제

```
class Counter {
    int count = 0;

    void increment() {
        count++;
    }
}
```

- 여러 스레드가 동시에 `increment()` 호출 → `count++` 연산이 **비원자적**
- 최종 결과가 예상보다 작음 (데이터 손실)

🟩 4. `synchronized` 사용한 해결

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}
```

- 이제 하나의 스레드만 `increment()`에 접근 가능 → **정상 동작**

🧩 5. 주의점

항목	설명
락 경합(lock contention)	여러 스레드가 같은 락을 기다려 **성능 저하** 발생 가능
데드락	서로 다른 락을 잡고 서로를 기다리는 상황 발생 가능
락 분리 전략	공유 자원이 다르다면 별도의 락 객체를 사용해 동시성 향상
`volatile` vs `synchronized`	`volatile`은 **가시성 보장**, `synchronized`는 **원자성 + 가시성 + 순서 보장**

🧠 정리

사용 대상	락 대상	의미
`synchronized method`	`this` or `Class.class`	전체 메서드 락
`synchronized block`	명시된 객체	블록 내부만 락

`wait()`, `notify()`, `notifyAll()`

🟦 1. 목적

- 여러 스레드가 **공유 객체**에 대해 **순서를 정해 작업**할 때 사용
- 예: 생산자-소비자 문제 (Producer-Consumer), 요청/응답 구조 등

🟧 2. 메서드 요약

메서드	설명
`wait()`	현재 스레드를 **대기 상태(WAITING)** 로 만들고 락을 **반납함**
`notify()`	대기 중인 스레드 중 **하나**를 깨움 (락은 여전히 현재 스레드가 보유)

메서드	설명
`notifyAll()`	대기 중인 **모든** 스레드를 깨움

> **주의:** 깨어난 스레드는 곧바로 실행되지 않음. 다시 락을 **획득해야** 실행됨.

🟥 3. 사용 조건

- 호출 대상은 `Object` (모든 Java 객체)
- 반드시 **동기화된(synchronized)** 블록 또는 메서드 안에서만 사용 가능
- 그렇지 않으면 `IllegalMonitorStateException` 예외 발생

🧩 4. 기본 사용 예제

```
class SharedObject {
    private boolean available = false;

    public synchronized void produce() throws InterruptedException {
        while (available) {
            wait();  // 소비가 끝날 때까지 대기
        }
        System.out.println("Produced");
        available = true;
        notify();  // 소비자 하나 깨움
    }

    public synchronized void consume() throws InterruptedException {
        while (!available) {
            wait();  // 생산이 끝날 때까지 대기
        }
        System.out.println("Consumed");
        available = false;
        notify();  // 생산자 하나 깨움
    }
}
```

- `wait()`은 **조건을 충족할 때까지 반복 확인 (while)** 해야 안전
- 깨어나더라도 **스풀링/경쟁** 때문에 조건을 재확인해야 함

🟩 5. `wait()` vs `sleep()` 차이

항목	`wait()`	`sleep()`
락 반납 여부	**락 반납함**	락 **유지함**
위치 제한	`synchronized` 내에서만 사용	어디서든 사용 가능

항목	`wait()`	`sleep()`
목적	스레드 간 협력	단순 지연

🧠 요약

- `wait()` → 현재 스레드를 일시 정지 + 락 반납
- `notify()` → 대기 스레드 하나를 깨움 (락은 여전히 점유)
- `notifyAll()` → 모든 대기 스레드를 깨움 (경쟁 상황 발생 가능)
- 반드시 `synchronized` 블록 내에서 사용해야 함

`Thread.sleep()`, `yield()`, `join()`

🟦 1. `Thread.sleep(long millis)`

✅ 개요

- 현재 실행 중인 스레드를 **지정된 시간만큼 일시 정지**시킴
- **CPU를 양보하고**, 일정 시간 후 다시 실행 가능 상태로 돌아감

✅ 특징

- `InterruptedException` 예외 처리 필요
- **락은 유지**한 채로 잠든다 → synchronized 블록 내 사용 시 주의

✅ 예시

```
System.out.println("Start");
Thread.sleep(1000);  // 1초 대기
System.out.println("End");
```

🟧 2. `Thread.yield()`

✅ 개요

- **현재 스레드가 CPU를 양보(yield)** 하고, **동등한 우선순위**의 다른 스레드에게 실행 기회를 줌
- 반드시 다른 스레드가 실행된다는 **보장은 없음**

✅ 특징

- 스케줄러에게 힌트를 주는 수준 (OS가 무시할 수도 있음)
- 일반적으로 거의 쓰이지 않지만, **busy wait 회피**나 **테스트 용도**로 사용됨

✅ 예시

```
while (true) {
    if (needBreak) break;
    Thread.yield();  // 잠깐 멈추고 다른 스레드에 기회 줌
}
```

🟥 3. `Thread.join()`

✅ 개요

- **다른 스레드가 종료될 때까지** 현재 스레드를 대기시킴
- 스레드 간 **작업 순서 제어**에 매우 유용

✅ 특징

- `join()`을 호출한 스레드는, 대상 스레드가 **종료될 때까지 블로킹**
- `join(long millis)`로 타임아웃 설정 가능
- `InterruptedException` 예외 처리 필요

✅ 예시

```
Thread t = new Thread(() -> {
    System.out.println("작업 중...");
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {}
    System.out.println("작업 끝!");
});

t.start();
t.join();  // main 스레드는 t 스레드가 끝날 때까지 기다림
System.out.println("모든 작업 종료");
```

🧠 차이 정리

메서드	설명	대상	예외 처리	락 해제 여부
`sleep()`	일정 시간 대기	현재 스레드	`InterruptedException`	❌ (락 유지)
`yield()`	실행 양보	현재 스레드	X	❌
`join()`	다른 스레드 종료까지 대기	다른 스레드	`InterruptedException`	❌

🧩 팁

- `sleep()` 은 CPU 부하를 줄이는 데 좋지만 락을 점유할 수 있음 → `wait()` 와의 구분 중요
- `yield()` 는 정확한 제어가 어려워 일반적인 실무에서는 거의 사용 안 함
- `join()` 은 **스레드 종료를 기다릴 때 매우 유용**, 단 병목 주의

`ExecutorService`와 쓰레드풀

`ExecutorService`는 자바에서 **스레드풀(thread pool)** 기반으로 **비동기 작업을 관리하고 실행**할 수 있게 해주는 고수준 API이다. 스레드를 직접 생성하고 관리하는 번거로움 없이, 효율적으로 작업을 처리할 수 있다.

🟦 1. 왜 `ExecutorService`를 쓰는가?

- `new Thread()`로 직접 스레드를 만들면:
  - 작업마다 새 스레드 생성 → 성능 저하
  - 스레드 개수 조절 불가 → **자원 낭비**
- `ExecutorService`는:
  - **스레드 재사용**
  - **작업 큐잉, 제한적 실행, 취소 등 제어 가능**
  - **비동기 처리**, **결과 추적**(Future)

🟧 2. 기본 구조

```
ExecutorService executor = Executors.newFixedThreadPool(3);  // 스레드 3개

executor.submit(() -> {
    System.out.println("비동기 작업 실행");
});

executor.shutdown();  // 더 이상 작업 받지 않음
```

🟩 3. 주요 구현 클래스 (`Executors` 팩토리 메서드)

메서드	설명
`Executors.newFixedThreadPool(n)`	고정 크기 스레드풀 생성
`Executors.newCachedThreadPool()`	작업 수에 따라 스레드 무한 생성 (단, 유휴 스레드는 제거됨)
`Executors.newSingleThreadExecutor()`	스레드 하나만 사용하는 작업 큐
`Executors.newScheduledThreadPool(n)`	지연 실행 또는 주기적 실행 지원

🟥 4. 주요 메서드

메서드	설명
`submit(Runnable)`	비동기 실행, 결과는 없지만 `Future` 반환
`submit(Callable)`	결과가 있는 비동기 실행, `Future<V>` 반환
`shutdown()`	더 이상 작업 안 받고, 현재 작업 완료 후 종료
`shutdownNow()`	즉시 실행 중인 스레드 인터럽트 시도
`awaitTermination()`	종료 대기
`invokeAll(List<Callable>)`	여러 작업을 동시에 실행하고, 모두 완료될 때까지 대기
`invokeAny(List<Callable>)`	여러 작업 중 **하나라도 완료되면** 그 결과를 리턴

🧩 예제: Callable + Future

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

public class ExecutorExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Callable<Integer> task = () -> {
            Thread.sleep(1000);
            return 42;
        };

        Future<Integer> future = executor.submit(task);
        System.out.println("작업 제출 완료");

        Integer result = future.get();  // 결과 받을 때까지 블로킹
        System.out.println("결과: " + result);

        executor.shutdown();
    }
}
```

🔁 스레드풀 동작 흐름

1. 작업이 제출되면 내부 작업 큐에 저장
2. 대기 중인 스레드가 꺼내서 실행
3. 스레드 수 부족 시, 대기하거나 (고정형) 새로 생성 (캐시형)
4. 작업이 끝나면 스레드는 **풀에 반납되어 재사용됨**

🧠 실무 팁

- `shutdown()` 을 **반드시 호출**해야 자원 누수 방지됨
- 스레드 수 조절은 시스템 코어 수 기준 설정 권장 (`Runtime.getRuntime().availableProcessors()`)
- `ThreadPoolExecutor` 를 직접 사용할 경우:
    - 큐 타입 (LinkedBlockingQueue 등), 거부 정책, keepAlive 설정 등 세부 조정 가능