EJ

이펙티브 자바 CP.10 동시성

작성 일시	@2023년 3월 25일 오후 9:04
최종 편집 일시	@2023년 5월 19일 오전 3:39
유형	이펙티브 자바
작성자	종현 박
참석자	
언어	

10 동시성

▼ 78. 공유 중인 가변 동기화해 사용하라

- `synchronized` 키워드 (동기화)
  - 해당 메서드나 블록을 한번에 한 스레드씩 수행하도록 보장한다.
  - 동기화를 배타적 실행, 즉 한 스레드가 변경하는 중이라서 상태가 일관되지 않은 순간의 객체를 다른 스레드가 보지 못하게 막는 용도.
  - 한 객체가 일관된 상태를 가지고 생성되고 (아이템 17), 이 객체에 접근하는 메서드는 그 객체에 락을 건다. 락을 건 메서드는 객체의 상태를 확인하고 필요하면 수정한다. 즉, 객체를 하나의 일관된 상태에서 다른 일관된 상태로 변화시킨다. 동기화를 제대로 사용하면 어떤 메서드도 이 객체의 상태가 일관되지 않는 순간을 볼 수 없을 것이다.
  - 동기화 없이는 한 스레드가 만든 변화를 다른 스레드에서 확인하지 못할 수 있다. **동기화는 일관성이 깨진 상태를 볼 수 없게하는 것은 물론, 동기화된 메서드나 블록에 들어간 스레드가 같은 락의 보호하에 수행된 모든 이전 수정의 최종 결과를 보게 해준다.**
- 원자성
  - 언어 명세상 `long`, `double` 외의 변수를 읽고 쓰는 동작은 원자적이라고 한다.
  - 여러 스레드가 같은 변수를 동기화 없이 수정하는 중이라도, 항상 어떤 스레드가 정상적으로 저장한 값을 온전히 읽어옴을 보장하는 뜻이다.
  - 하지만, 원자적 데이터를 읽고 쓸때는 동기화를 하지 말아야 겠다는 생각은 아주 위험한 생각이다. **자바 언어 명세는 스레드가 필드를 읽을 때 항상 수정이 완전히 반영된 값을 얻는다고 보장하지만. 한 스레드가 저장한 값이 다른 스레드에게 보이는지는 보장하지 않는다.**
    - 이는 한 스레드가 만든 변화가 다른 스레드에게 언제 어떻게 보이는지를 규정한 자바의 메모리 모델 때문
  - **동기화는 배터적 실행뿐 아니라, 스레드 사이의 안정적인 통신에 꼭 필요하다.**
- 동기화 실패
  - 공유 중인 가변 데이터를 비록 원자적으로 읽고 쓸 수 있을지라도 동기화에 실패하면 처참한 결과로 이어질 수 있다.
  - `Thread.stop` 데이터가 훼손 될 수 있다. (사용 금지) - deprecated 되었다. (1.2)

- 올바르게 스레드를 멈추는 코드
  - 첫 번째 스레드는 자신의 `boolean` 필드를 폴링하면서 그 값이 `true` 가 되면 멈춘다.
  - 이 필드를 `false` 로 초기화 해두고, 다른 스레드에서 이 스레드를 멈추고자 할 때 `true` 로 변경하는 식이다.
  - ex) 잘못된 방식 - `boolean` 필드가 원자적이라 동기화를 안 한 경우

```
public class StopThread {
    private static boolean stopRequested;

    public static void main(String [] args) throw InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

    - 메인 스레드가 1초 후 `stopRequested` 를 `true` 로 설정하는 `backgroundThread` 는 반복문을 빠져나올 것 처럼 보이지만, 영원히 수행된다.
    - 동기화하지 않으면 메인 스레드가 수정한 값을 백그라운드 스레드가 언제쯤에나 보게될지 보장할 수 없다.
    - **호이스팅 (끌어올리기)**
      - OpenJDK 서버 VM 이 실제로 호이스팅이라는 최적화 기법으로 해당 코드는 이렇게 변경된다.

```
// before
while (!stopRequest)
    i++;

// after
if (!stopRequested)
    while (true)
        i++;
```

      - 즉, 이 프로그램은 **응답 불가** 상태가 되어 더 이상 진전이 없다.
  - ex) 올바른 방법 `synchronized` 사용

```
public class StopThread {
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String [] args) throw InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (! stopRequested())
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

    - 쓰기, 읽기 모두 동기화되지 않으면 동작을 보장하지 않는다. 반드시 쓰기, 읽기을 `synchronized` 통해 가변 데이터를 동기화 하자. (통신 목적으로 사용된 것)

- ex) 올바르고 더 속도가 빠른 대안 `volatile` 선언해서 동기화

```
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String [] args) throw InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

- `volatile` **한정자 (통신 목적)**
  - 배터적 수행과는 상관 없지만 항상 가장 최근에 기록된 값을 읽게 됨을 보장한다.
  - 하나의 스레드에서만 쓰기 작업, 나머지 여러 쓰레드에서 읽기 작업을 보장한다
  - 하지만, 주의해서 사용해야한다.
    - ex) 일련번호 생성

```
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

      - 이 메서드는 매번 고유한 값을 반환할 의도로 만들어졌다.
      - 이 메서드의 상태는 `nextSerialNumber` 라는 필드로 결정되는데, 원자적으로 접근할 수 있고 어떤 값이든 허용한다. 따라서 굳이 동기화하지 않아도 불변식을 보호할 수 있어 보인다. (하지만, 이역시 동기화 없이는 올바로 동작하지 않는다.)
      - 문제는 증가 연산자(++) 다.
        이 연산자는 코드상으로는 하나지만, 실제로는 `nextSerialNumber` 에 두 번 접근 한다. 먼저 값을 읽고, 그런 다음 새로운 값을 저장하는 것이다.
      - 만약 **두 번째 스레드가 이 두 접근 사이를 비집고 들어와 값을 읽어가면 첫 번째 스레드와 똑같은 값을 돌려받게 된다. 이런 오류를 안전 실패라고 한다.**
        - 이런 문제를 해결하기 위해 `synchrnized` 한정자를 붙이고 `volatile`을 제거해야한다.
- **atomic 패키지**
  - 동시성 프로그래밍을 위한 여러 자바 표준 기술들을 제공하는데 락 없이도 스레드 안전한 프로그래밍들을 지원하는 클래스들이 다양하게 존재하고, `volatile` 과 다르게 안전 통신 목적 뿐만 아니라, 배타적 실행 모두 제공하면서 성능도 좋다
  - ex) `AtomicLong` 사용

```
private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {
    return nextSerialNum.getAndIncrement();
}
```

- **가장 좋은 방법은 애초에 가변 데이터를 공유하지 않는 것이다.**
  - 불변 데이터 (아이템 17)만 공유하거나 아무것도 공유하지 말자.
    가변 데이터는 단일 스레드에서만 쓰도록 하자.

    - 한 스레드가 데이터를 다 수정한 후 다른 스레드에 공유할 때는 해당 객체에서 공유하는 부분만 동기화해도 된다. 그러면 그 객체를 다시 수정할 일이 생기기 전까지 다른 스레드들은 동기화 없이 자유롭게 값을 읽어갈 수 있다. 이런 객체를 사실상 **불변** 이라 하고 다른 스레드에 이런 객체를 건네는 행위를 **안전 발행**이라 한다.
        - 객체를 안전하게 발행하는 방법으로는 초기화 과정에서 객체를 정적 필드, volatile 필드, final 필드, 보통의 lock 을 통해 접근 하는 필드에 저장 등이 있다.
- 정리
    - 여러 스레드가 가변 데이터를 공유하면 그 데이터를 읽고 쓰는 동작은 반드시 동기화를해야한다.
    - `synchrnized` , `atomic`패키지 을 사용하면, 배타적 실행과 안전한 통신을 가능하게 할 수 있다.
    - `volatile` 을 사용하면 안전한 통신만 가능하게 할 수 있다.

▼ 79. 과도한 동기화는 피하라

- 동기화의 단점
    - 과도한 동기화는 성능을 떨어진다.
    - 교착 상태에 빠뜨리고, 심지어 예측하지 못하는 동작을 낳을 수 있다.
- **응답 불가와 안전 실패를 피하려면 동기화 메서드나 동기화 블록 안에서는 제어를 절대로 클라이언트에 양도하면 안 된다.**
    - 동기화 영역 안에서는 재정의할 수 있는 메서드는 호출하면 안 된다.
    - 클라이언트가 넘겨준 함수 객체(람다, 익명 함수)(아이템 24)를 호출해서도 안 된다.
    - 위 같은 메서드를 **외계인 메서드**라고 한다.
        - 동기화된 영역을 포함한 클래스 관점에서는 해당 메서드가 무슨 일을 할지 알지 못하며 통제할 수도 없다는 뜻이다.
        - 하는 일에 따라 동기화된 영역은 예외를 일으키거나, 교착상태에 빠지거나, 데이터를 훼손 할 수 있다.
        - 얼마나 오래 실행될지 알 수 없는데, 동기화 영역 안에서 호출된다면 그동안 다른 스레드는 보호된 자원을 사용하지 못하고 대기해야만 한다.
    - ex) 잘못된 코드, 동기화 블록 안에서 외계인 메서드를 호출 (ForwardingSet)

```
public class ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers = new ArrayList<>();

    public void addObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.add(observer);
        }
    }

    public void removeObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.remove(observer);
        }
    }

    private void notifyElementAdded(E element) {
        synchronized(observers) {
            for (SetObserver<E> observer : observers)
                observer.added(this, element);
        }
    }

    @Override
    public boolean add(E element) {
        boolean added = super.add(element);
        if (added)
            notifyElementAdded(element);
        return added;
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        boolean result = false;
        for (E element : c)
            result |= add(element);
```

```
        return result;
    }
}
```

- `Observer` 들은 `addObserver` , `removeObserver` 메서드를 호출해 구독을 신청하거나 해지한다. 두 경우 모두 다음 콜백 인터페이스의 인스턴스를 메서드에 건넨다.

```
@FunctionalInterface
public interface SetObserver<E> {
    // ObservableSet에 원소가 더해지면 호출된다.
    void added(ObservableSet<E> set, E element);
}
```

  - 눈으로 볼 때, 해당 `ObservableSet` 은 잘 동작할 것 같다.

```
public static void main(String[] args) {
    ObservableSet<Integer> set = new ObservableSet<>(new HashSet<>());

    set.addObserver((s,e) -> System.out.println(e));

    for (int i = 0 ; i < 100; i++)
        set.add(i);
}
```

    - 0 ~ 99 까지 잘 출력한다.

  - 하지만, 어떤 값일 때 자기 자신을 제거(구독 해지) 하는 관찰자로 변경하면 - **예1**

```
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23)
            s.removeObserver(this);
    }
});
```

> 람다를 사용한 이전 코드와 달리 익명 클래스를 사용
> `s.removeObserver` 메서드에 함수 객체 자신을 넘겨야 하기 때문이다.
> 람다는 자신을 참조할 수단이 없다. (아이템 42)

    - 이 프로그램은 0 ~ 23까지 출력한 후 관찰자 자신을 구독해지한 다음 조용히 종료될 것으로 보인다.
      하지만, 실제 실행시, 이 프로그램은 23까지 출력 후 `ConcurrentModificationException` 을 던진다. 관찰자의 `added` 메서드 호출이 일어난 시점이 `notifyElementAdded` 가 관찰자들의 리스트를 순회하는도중이기 때문이다.
    - `added` 메서드는 `ObservableSet` 의 `removeObserver` 메서드를 호출하고, 이 메서드는 다시 `observers.remove` 메서드를 호출한다.
      여기서 문제가 발생한다. 리스트에서 원소를 제거하려 하는데, 마침 지금은 이 리스트를 순회하는 도중이다. 즉, 허용되지 않은 동작이다.
      `notifyElementAdded` 메서드에서 수행하는 순회는 동기화 블록 안에 있으므로 동시 수정이 일어나지 않도록 보장하지만, 정작 자신이 콜백을 거쳐 되돌아와 수정하는 것까지 막지 못한다.

- `removeObserver` 를 직접 호출하지 않고 실행자 서비스 (아이템 80)을 사용해 다른 스레드에게 부탁하는 예제 - **예2**

```
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) {
            ExecutorService exec = Executors.newSingleThreadExecutor();
            try {
                exec.submit(() -> s.removeObserver(this)).get();
            } catch (ExecutionException | InterruptedException ex) {
```

```
                throw new AssertionError(ex);
            } finally {
                exec.shutdown();
            }
        }
    }
});
```

    - 이 프로그램은 실행하면 예외는 나지 않지만, 교착 상태에 빠진다.
      백그라운드 스레드가 `s.removeObserver` 를 호출하면 관찰자를 잠그려 시도하지만 락을 얻을 수 없다. 메인 스레드가 이미 락을 쥐고 있기 때문이다.
      그와 동시에 메인 스레드는 백그라운드 스레드가 관찰자를 제거하기만을 대기하는 중이다. 바로 **교착상태**..
    - 같은 상황이지만, 불변식이 임시로 깨진 경우
      자바 언어의 락은 재진입을 허용하므로 교착상태에 빠지지 않는다. 예외를 발생시킨 첫 번째 예에서라면 외 계인 메서드를 호출하는 스레드는 이미 락을 쥐고 있으므로 다음번 락 획득도 성공한다. 그 락이 보호하는 데이터에 대해 개념적으로 관련이 없는 다른 작업이 진행 중인데도 말이다.
    - 이런 재진입 가능 락(ReentrantLock)은 객체 지향 멀티스레드 프로그램을 쉽게 구현할 수 있게 해주지만, 응답 불가(교착 상태)가 될 상황을 안전 실패(데이터 훼손)로 변모 시킬 수 있다.

- 해결 방법

  1. 외계인 메서드 호출을 동기화 블록 바깥으로 옮기면 된다.
     (동기화 영역 바깥에서 호출되는 외계인 메서드를 **열린 호출** 이라 한다.)

     - `notifyElementAdded` 메서드라면 관찰자 리스트를 복사해 쓰면 락 없이도 안전히 순회 가능하다. (예외 발생과 교착상태 증상이 사라 질 것이다.)

```
private void notifyElementAdded(E element) {
    List<SetObserver<E>> snapshot = null;
    synchronized(observers) {
        snapshot = new ArrayList<>(observers);
    }
    for (SetObserver<E> observer : snapshot)
        observer.add(this, element);
}
```

  2. 자바의 동시성 컬렉션 라이브러리의 `CopyOnWriteArrayList` 가 정확히 이 목적으로 특별히 설계됐다.

     - 내부를 변경하는 작업은 항상 깨끗한 복사본을 만들어 수행하도록 구현
     - 내부 배열은 절대 수정되지 않으니, 순회시 락이 필요 없어 매우 빠름

```
private final List<SetObserver<E>> observers = new CopyOnWriteArrayList<>();

public void addObserver(SetObserver<E> observer) {
    observers.add(observer);
}

public boolean removeObserver(SetObserver<E> observer) {
    return observers.remove(observer);
}

private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
        observer.added(this, element);
}
```

- 동기화 기본 규칙은 동기화 영역에서 가능한 한 일을 적게 하는 것이다.
  - 락을 얻고, 공유 데이터를 검사하고, 필요하면 검사하고, 락을 놓는다.
  - 오래 걸리는 작업이라면 아이템 78의 지침을 어기지 않으면서 동기화 영역 바깥으로 옮기는 방법을 찾아보자.
- 가변 클래스 작성
  1. 동기화를 전혀 하지 말고, 그 클래스를 동시에 사용해야 하는 클래스가 외부에서 알아서 동기화하게 하자.

        - `java.util`
    2. 동기화를 내부에서 수행해 스레드 안전한 클래스로 만들자 (아이템 82)
       단, 클라이언트가 외부에서 객체 전체에 락을 거는 것보다 동시성을 월등히 개선할 수 있을 때만, 두 번째 방법을 선택해야 한다.
        - `java.util.concurrent`
        - `StringBuffer`, `ThreadLocalRandom`
        - 락 분할, 락 스트라이핑, 비차단 동시성 제어 등 다양한 기법을 동원해 동시성을 올려 줄 수 있다.
- 정리
    - 교착상태와 데이터 훼손을 피하려면 동기화 영역 안에서 외계인 메서드를 절대 호출하지 말자.
    - 동기화 영역 안에서의 작업은 최소한으로 줄이자.
    - 가변 클래스를 설계할 때는 스스로 동기화해야 할지 고민하자.
    - 합당한 이유가 있을 때만, 내부에서 동기화하고, 동기화 내부 구현 여부를 문서에 밝히자.

▼ 80. 스레드보다는 실행자, 태스크, 스트림을 애용하라

- `java.util.concurrent` 패키지
    - 실행자 프레임워크라고 하는 인터페이스 기반의 유연한 테스크 실행 기능을 담고 있다.

```
ExecutorService exce = Executors.newSingleThreadExecutor();
// 실행할 테스크를 넘기는 방법
exec.execute(runnable);
// 실행자를 종료시키는 방법
exec.shutdown();
```

    - 주요 기능 (여길 참고 했다.)
        - 특정 태스크가 완료되기를 기다린다.

```
ExecutorService exec = Executors.newSingleThreadExecutor();
exec.submit(()  -> s.removeObserver(this)).get(); // 끝날 때까지 기다린다.
```

        - 태스크 모음 중 아무것 하나(`invokeAny` 메서드) 혹은 모든 태스크(`invoke all` 메서드)가 완료되길 기다린다.

```
List<Future<String>> futures = exec.invokeAll(tasks);
System.out.println("All Tasks done");

exec.invokeAny(tasks);
System.out.println("Any Task done");
```

        - 실행자 서비스가 종료하기를 기다린다. (`awaitTermination` 메서드)

```
Future<String> future = exec.submit(task);
exec.awaitTermination(10, TimeUnit.SECONDS);
```

        - 완료된 태스크들의 결과를 차례로 받는다. (`ExecutorCompletionService` 이용)

```
final int MAX_SIZE = 3;
ExecutorService executorService = Executors.newFixedThreadPool(MAX_SIZE);
ExecutorCompletionService<String> executorCompletionService = new ExecutorCompletionService<>(executorService);

List<Future<String>> futures = new ArrayList<>();
futures.add(executorCompletionService.submit(() -> "madplay"));
futures.add(executorCompletionService.submit(() -> "kimtaeng"));
futures.add(executorCompletionService.submit(() -> "hello"));

for (int loopCount = 0; loopCount < MAX_SIZE; loopCount++) {
    try {
        String result = executorCompletionService.take().get();
```

```
            System.out.println(result);
        } catch (InterruptedException e) {
            //
        } catch (ExecutionException e) {
            //
        }
    }
    executorService.shutdown();
```

    - 태스크를 특정 시간 혹은 주기적으로 실행하게 한다. (`ScheduledThreadPoolExecutor` 이용)

```
ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(1);

executor.scheduleAtFixedRate(() -> {
    System.out.println(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")
            .format(LocalDateTime.now()));
}, 0, 2, TimeUnit.SECONDS);

// 2019-09-30 23:11:22
// 2019-09-30 23:11:24
// 2019-09-30 23:11:26
// 2019-09-30 23:11:28
// ...
```

- 스레드 풀
    - 스레드 풀의 스레드 개수는 고정할 수도 있고 필요에 따라 늘어나거나 줄어들게 설정할 수 있다.
    - 필요한 실행자 대부분은 `java.util.concurrent.Executors`의 정적 팩토리들을 이용해 생성할 수 있습니다.
    - 평범하지 않은 실행자를 원한다면, `ThreadPoolExecutor` 클래스를 직접 사용해도 된다.
- `Executors.newCachedThreadPool`
    - 작은 프로그램이나 작은 서버에 유용하다.
    - 특별히 설정할 게 없고 일반적인 용도에 적합하게 동작한다.
    - 하지만, 무거운 프로덕션 서버에는 좋지 못하다.
        - 요청받은 태스크들이 큐에 쌓이지 않고 즉시 스레드에 위임돼 실행되고, 가용한 스레드가 없다면 새로 하나 생성한다. 서버는 CPU 이용률 100%에 치닫고, 새로운 태스크가 도착하는 족족 또 다른 스레드를 생성하여 상황을 악화시킨다.
        - 따라서 무거운 프로덕션 서버에서는 `Executors.newFixedThreadPool` 을 선택하거나 완전히 통제할 수 있는 `ThreadPoolExecutor` 를 직접 사용하는게 낫다.
- 스레드를 직접 다루지 말자.
    - 스레드를 직접 다루면 Thread 가 작업 단위와 수행 매커니즘 역할을 모두 수행하게 된다.
    - 반면 실행자 프레임워크에서는 작업 단위와 실행 매커니즘이 분리된다.
    - 작업 단위를 나타내는 핵심 추상 개념이 태스크다.
    - 태스크의 종류
        1. `Runnable`
        2. `Callable` (`Runnable` 과 비슷하지만 값을 반환하고 임의의 예외를 던진다.)
    - 실행자 서비스
        - 스크를 수행하는 일반적인 매커니즘
        - 태스크 수행을 실행자 서비스에 맡기면 원하는 태스크 수행 정책을 선택할 수 있고, 생각이 바뀌면 언제든 변경할 수 있다.
        - 핵심은 실행자 프레임워크가 작업 수행을 담당해준다는 것이다.
    - 포크 조인 태스크 (`ForkJoinTask`)
        - 자바 7 부터 실행자 프레임 워크는 포크 조인 태스크를 지원하도록 확장됨.

    - 포크 조인 태스크는 포크 조인 풀이라는 특별한 실행자 서비스가 실행해준다.
    - 포크 조인 태스크의 인스턴스는 작은 하위 태스크로 나뉠 수 있고, `ForkJoinPool` 을 구성하는 스레드들이 이 태스크들을 처리하며, 일을 먼저 끝낸 스레드는 다른 스레드의 남은 태스크를 가져와 대신 처리할 수도 있다.
      이렇게 하여 모든 스레드가 바쁘게 움직여 CPU를 최대한 활용하고 높은 처리량과 낮은 지연시간을 달성한다.

▼ 81. wait 와 notify 보다는 동시성 유틸리티를 애용하라

- `notify, wait`
    - `notify` - 일시 정지 상태인 다른 스레드를 실행 대기 상태로 만듦
    - `wait` - 스레드를 일시 정지 상태로 만듦
    - Java 5 이후라면 동시성 유틸리티가 지원함으로
      `wait` 와 `notify` 는 올바르게 사용하기가 아주 까다로우니 고수준 동시성 유틸리티를 사용하자.
- `java.util.concurrent` 의 고수준 유틸리티
    1. 실행자 프레임워크 - (아이템 80)
    2. 동시성 컬렉션 (`concurrent collection`)
        - `List`, `Queue`, `Map` 표준 컬렉션 인터페이스에 동시성을 추가해 구현한 고성능 컬렉션이다.
        - 높은 동시성에 도달하기 위해 동기화를 각자의 내부에서 수행한다. (아이템 79)
        - **따라서, 동시성 컬렉션에서 동시성을 무력화하는 건 불가능하며, 외부에서 락을 추가로 사용하면 오히려 속도가 느려진다.**
        - 동시성 컬렉션에서 동시성을 무력화하지 못하므로 여러 메서드를 원자적으로 묶어 호출하는 일 역시 불가능하다.
          그래서 여러 기본 동작을 원자적 동작으로 묶는 '상태 의존성 수정' 메서드들이 추가되었다.
        - ex) `Map` 의 `putIfAbsent(key, value)`
            - 주어진 키에 매핑된 값이 아직 없을 때만 새 값 집어넣는다.
            - 그리고 주어진 값이 있다면 그 값을 반환하고 없다면 `null` 을 반환한다.
            - ex) `String.intern` 동작을 흉내 내어 구현한 메서드

```
private static final ConcurrentMap<String, String> map = new ConcurrentHashMap<>();

// 동시성 정규화 맵 - 최적은 아님
public static String intern(String s)
    String previousValue = map.putIfAbsent(s,s);
    reutrn previousValue == null ? s : previousValue;
}

// 동시성 정규화 맵 - 더 빠름
// get 을 먼저 호출하여 필요할 때만 putIfAbsent 호출한다.
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s,s);
        if (result == null)
            result = s;
    }
    return result;
}
```

            - `ConcurrentHashMap` 은 동시성이 뛰어나며 속도도 빠르다.
            - 동기화된 맵을 동시성 맵으로 교체하는 것만으로도 동시성 애플리케이션의 성능은 극적으로 개선된다.
        - 컬렉션 인터페이스 중 일부는 작업이 성공적으로 완료될 때까지 기다리도록(차단되도록) 확장되었다.
            - ex) Queue 확장한 BlockingQueue 에 추가된 메서드 중 take는 큐의 첫 원소를 꺼낸다. 이때 만약 큐가 비었다면 새로운 원소가 추가될 때까지 기다린다. 이런 특성 덕에 BlockingQueue 는 작업 큐(생산자-소비자 큐)로 쓰기에 적합하다.
              작업 큐는 하나 이상의 생산자와 스레드가 작업을 큐에 추가하고, 하나 이상의 소비자 스레드 큐에 있는 작업을 꺼내 처리하는 형태이다.

3. 동기화 장치 (`synchronizer`)
    - 스레드가 다른 스레드를 기다릴 수 있게 하여, 서로 작업을 조율할 수 있게 해준다.
    - 가장 자주 쓰이는 동기화 장치는 `CountDownLatch` 와 `Semaphore` 다.
      `CyclicBarrier` 와 `Exchanger` 는 그보다 덜 쓰인다.
      가장 강력한 동기화 장치는 바로 `Phaser` 다
    - `CountDownLatch` 는 일회성 장벽으로, 하나 이상의 스레드가 또 다른 하나 이상의 스레드 작업이 끝날 때까지 기다리게 한다.
      유일한 생성자는 int 값을 받으며, 이 값이 래치의 `countDown` 메서드를 몇번 호출해야 대기 중인 스레들을 깨우는지를 결정한다.
    - ex) `CountDownLatch` 를 사용해 동시 실행 시간을 재는 프레임워크

```
public static long time(Executor executor, int concurrency, Runnable action) throws InterruptedException {
    CountDownLatch ready = new CountDownLatch(concurrency);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done = new CountDownLatch(concurrency);

    for (int i = 0; i < concurrency; i++) {
        executor.execute(() -> {
            // 타이머에게 준비를 맞쳤음을 알린다.
            ready.countDown();
            try {
                // 모든 작업자 스레드가 준비될 때까지 기다린다.
                start.await();
                action.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                // 타이머에게 작업을 마쳤음을 알린다.
                done.countDown();
            }
        });
    }

    ready.await(); // 모든 작업자가 준비될 때까지 기다린다.
    long.startNanos = System.nanoTime();
    start.countDown(); // 작업자들을 깨운다.
    done.await(); // 모든 작업자가 일을 끝마치기를 기다린다.
    return System.nanoTime() - startNanos;
```

        - 이 메서드는 동작들을 실행할 실행자와 동작을 몇 개나 동시에 수행 할 수 있는지를 뜻하는 동시성 수준 (`concurrency`) 를 매개변수로 받는다.
        - 타이머 스레드가 시계를 시작하기 전에 모든 작업자 스레드는 동작을 수행할 준비를 마친다. 마지막 작업자 스레드가 준비를 마치면 타이머 스레드가 '시작 방아쇠(`start.countDown()`)' 를 당겨 작업 스레드들이 일을 시작하게 한다. 마지막 작업 스레드가 동작을 마치자마자 타이머 스레드는 시계를 멈춘다.
        - `ready` 래치는 작업자 스레드들이 준비가 완료됐음을 타이머 스레드에 통지할 때 사용된다.
          통지를 끝낸 작업자 스레드들은 두 번째 래치인 `start` 가 열리길 기다린다.
          마지막 작업자 스레드가 `ready.countDown` 을 호출하면 타이머 스레드가 시작 기록을 기록하고 `start.countDown` 을 호출하여 기다리던 작업자들을 깨운다.
          그 직후 타이머 스레드는 세번째 래치인 `done` 이 열리기를 기다린다. `done` 래치는 마지막 남은 작업자 스레드 동작을 마치고 `done.countDown` 을 호출하면 열린다.
        - 세부사항
            - `time` 메서드에 넘겨진 실행자(`executor`) 는 `concurrency` 매개변수로 지정한 동시성 수준 만큼의 스레드를 생성할 수 있어야 한다. 그렇게 못하면 이 메서드는 결코 끝나지 않는다. 이런 상태를 **스레드 기아 교착 상태(thread starvation deadlock)** 라고 한다.
            - `InterruptedException` 을 캐지한 작업자 스레드는 `Thread.currentThread().interrupt()` 관용구를 사용해 인터럽트를 되살리고 자신은 run 메서드에서 빠져 나온다. 이렇게 해야 실행자가 인터럽트를 적절하게 처리할 수 있다.
            - 시간 간격을 잴 때는 항상 `System.currentTimeMillis` 가 아닌 `System.nanoTime` 을 사용하자. (더 정확하고 정밀하며 시스템의 실시간 시계의 보정에 영향을 받지 않음)

        - `CyclicBarrier` (혹은 `Phaser`) 인스턴스 하나로 대체하면, 코드가 명료해지지만, 이해하기가 어려울 것이다.

- 새로운 코드라면 언제나 `wait`, `notify` 가 아닌 동시성 유틸리티를 써야 한다.
  하지만 어쩔 수없이 해당 레거시 코드를 다뤄야 할 때, `wait` 메서드는 스레드가 어떤 조건이 충족되기를 기다리게 할 때 사용한다. 락 객체의 `wait` 메서드는 반드시 그객체를 잠근 동기화 영역 안에서 호출해야 한다.
  - ex) `wait` 메서드를 사용하는 표준 방식

```
synchronized (obj) {
    while (<조건이 충족되지 않았다>)
        obj.wait(); // 락을 놓고, 깨어나면 다시 잡는다.

    ... // 조건이 충족될 때의 동작을 수행한다.
}
```

  - `wait` 문을 사용할 때는 반드시 대기 반복문 (`wait loop`) 관용구를 사용하라. 반복문 밖에서는 절대로 호출하지 말자. 이 반복문은 `wait` 호출 전후로 조건이 만족하는지를 검사하는 역할을 한다.
  - 대기 전에 조건을 검사하여 조건이 이미 충족되었다면 `wait` 를 건너뛰게 한 것은 응답 불가 상태를 예방하는 조치이다. 만약 조건이 이미 충족되었는데 스레드가 `notify` (혹은 `notifyAll`) 메서드를 먼저 호출한 후 대기 상태로 빠지면, 그 스레드를 다시 깨울 수 있다고 보장할 수 없다.
  - 한편, 대기 후에 조건을 검사하여 조건이 충족되지 않았다면 다시 대기하는 것은 안전 실패를 막기 위한 조치이다. 만약 조건이 충족되지 않았는데 스레드가 동작을 이어가면 락이 보호하는 불변식을 깨뜨릴 위험이 있다.
  - 조건이 충족되지 않아도 스레드가 깨어 날 수 있는 상황
    - 스레드가 `notify` 를 호출한 다음 대기 중이던 스레드가 깨어나는 사이에 다른 스레드가 락을 얻어 그 락이 보호하는 상태로 변경한다.
    - 조건이 만족되지 않았음에도 다른 스레드가 실수 혹은 악의적으로 `notify` 를 호출한다. 공개된 객체를 락으로 사용해 대기하는 클래스는 이런 위험에 노출된다. 외부에 노출된 객체의 동기화된 메서드 안에서 호출하는 `wait` 는 모두 이 문제에 영향을 받는다.
    - 깨우는 스레드는 지나치게 관대해서, 대기 중인 스레드 중 일부만 조건이 충족되어도 `notifyAll` 을 호출해 모든 스레드를 깨울 수 도 있다.
    - 대기 중인 스레드가 `notify` 없어도 깨어나는 경우가 있다. 허위 각성 이라는 현상이다.
  - `notify` 는 스레드 하나만 깨우며, `notifyAll` 은 모든 스레드를 깨운다.
    - 일반적으로는 `notifyAll` 을 사용하는게 합리적이고 안전하다.
    - 깨어나야 하는 모든 스레드가 깨어남을 보장하니 항상 정확한 결과를 얻을 것이다. 다른 스레드에 영향이 갈 수도 있지만, 프로그램의 정확성에는 영향을 주지 않을 것임.
    - 외부로 공개된 객체에 대해 실수로 혹은 악의적으로 `notify` 를 호출하는 상황에 대비하기 위해 `wait` 를 반복문 안에서 호출했듯, `notify` 대신 `notifyAll` 을 사용하면 관련 없는 스레드가 실수 혹은 악의적으로 `wait` 를 호출하는 공격으로 부터 보호 받을 수 있다.
- 정리
  - `wait`, `notify` 를 직접 사용하는건 어셈블리 언어를 프로그래밍 하는 것으로 비유할 수 있다. (`java.util.concurrent` 고수준 언어)
  - 코드를 새로 작성한다면 `wait`, `notify` 를 사용하지 말고 `java.util.concurrent` 를 사용하자
  - 만약, `wait, notify` 를 사용한다면 `wait` 는 항상 `while` 문 안에서 호출하고 (표준 관용구). `notify` 보다는 `notifyAll` 을 사용하자.

▼ 82. 스레드 안전성 수준을 문서화 해라

- 개요
  - 여러 스레드에서 한 메서드를 동시에 호출할 때, 그 메서드가 멀티 스레드 환경에서는 어떻게 동작하는지 아무런 언급이 없으면, 그 클래스 사용자는 나름의 가정을 해야만 한다.

    - 즉 스레드 안전성 수준을 문서화 하자.
- `synchronized` 한정자만으로 스레드 안전하다고 믿을 수 없다.
    - javadoc 기본 옵션에서 생성한 API 문서에는 `synchronized` 한정자가 포함되지 않는다.
    - **메서드 선언에 `synchronized` 한정자를 선언할지는 구현 이슈일 뿐 API 에 속하지 않는다.**
    - 더구나 `synchronized` 유무로 스레드 안전성을 알 수 있다는 것은 “스레드 안전성이 모 아니면 도” 라는 오해에 뿌리 둔 것이다. 하지만 스레드 안전성에도 수준이 나뉜다.
      멀티스레드 환경에서도 API를 안전하게 사용하게 하려면 클래스가 지원하는 스레드 안전성 수준을 정확히 명시해야 한다.
- 스레드 안전성 수준 (안전성이 높은 순)
    - **불변(immutable)** - 이 클래스의 인스턴스는 마치 상수와 같아서 외부 동기화도 필요 없다. `String, Long, BigInteger` 등
    - **무조건적 스레드 안전(unconditionally thread-safe)** - 이 클래스의 인스턴스는 수정될 수 있으나, 내부에서 충실히 동기화하여 별도의 외부 동기화 없이 동시에 사용해도 안전하다. `AtomicLong, ConcurrentHashMap` 등
        - 비공개 락 객체 관용구 사용 가능
    - **조건부 스레드 안전(conditionally thread-safe)** - 무조건적 스레드 안전과 같으나, 일부 메서드는 동시에 사용하려면 외부 동기화가 필요하다. `Collection.synchronized` 래퍼 메서드가 반환한 컬렉션 (이 컬렉션들이 반환한 반복자는 외부에서 동기화해야 한다)
        - 상세 정보
    - **스레드 안전하지 않음(not thread-safe)** - 이 클래스의 인스턴스는 수정될 수 있다. 동시에 사용하려면 각각의 메서드 호출을 클라이언트가 선택한 외부 동기화 매커니즘으로 감싸야 한다. `ArrayList, HashMap` 같은 기본 컬렉션
    - **스레드 적대적(thread-hostile)** - 이 클래스는 모든 메서드 호출을 외부 동기화로 감싸더라도 멀티스레드 환경에서 안전하지 않다. 이 수준의 클래스는 일반적으로 정적 데이터를 아무 동기화 없이 수정한다. 이런 클래스를 고의로 만드는 사람은 없겠지만, 동시성을 고려하지 않고 작성하다 보면 우연히 만들어질 수 있다. 이런 클래스나 메서드는 일반적으로 문제를 고쳐 재배포 하거나 사용 자제(deprecated) API 로 등록된다.
- 조건부 스레드 안전 상세 정보
    - 조건부 스레드 안전 클래스는 주의해서 문서화해야 한다. 어떤 순서로 호출할 때 외부 동기화가 필요한지, 그리고 그 순서로 호출하려면 어떤 락 혹은 락들을 얻어야 하는지 알려줘야 한다. 일반적으로 인스턴스 자체를 락으로 얻지만 예외도 있다.
    - ex) `Collections.synchronizedMap` API 문서

      > **synchronizedMap이 반환한 맵의 컬렉션 뷰를 순회하려면 반드시 그 맵을 락으로 사용해 수동으로 동기화하라.**
      >
      > Map<K, V> m = Collections.synchronizedMap(new HashMap<>());
      > Set<K> s = m.keySet(); // 동기화 블록 밖에 있어도 된다.
      > …
      > synchronized(m) { // **s가 아닌 m 을 사용해 동기화 해야한다.**
      > for (K key : s)
      > key.f();
      > }
      >
      > 이대로 따르지 않으면 동작을 예측할 수 없다.
- 문서화 방법
    - 클래스의 스레드 안전성은 보통 클래스의 문서화 주석에 기재하지만, 독특한 특성의 메서드라면 해당 메서드의 주석에 기재하도록 하자. 열거 타입은 굳이 불변이라고 쓰지 않아도 된다. 반환 타입만으로는 명확히 알 수 없는 정적 팩터리라

면 자신이 반환하는 객체의 스레드 안전성을 반드시 문서화해야 한다. (위 `Collections.synchronizedMap` 좋은 예)

- 외부에 공개된 락
  - 클래스가 외부에서 사용할 수 있는 락을 제공하면 클라이언트에서 일련의 메서드 호출을 원자적으로 수행할 수 있다. 하지만 이 유연성에는 대가가 따른다.
  - 내부에서 처리하는 고성능 동시성 제어 매커니즘과 혼용할 수 없게 된다.
    그래서 `ConcurrentHashMap` 같은 동시성 컬렉션과는 함께 사용하지 못한다.
  - 클라이언트가 공개된 락을 오래 쥐고 놓지 않는 서비스 거부 공격을 수행할 수도 있다.
    - 서비스 거부 공격을 막으려면 `synchronized` 메서드 대신 비공개 락 객체를 사용해야 한다.
    - ex) 비공개 락 객체 관용구 - 서비스 거부 공격을 막아준다.

```
private final Object lock = new Object();

public void foo() {
    synchronized(lock) {
        ...
    }
}
```

      - 비공개 락 객체는 클래스 바깥에서는 볼 수 없으니 클라이언트가 그 객체의 동기화에 관여할 수 없다. 사실 아이템 15 조언에 따라 락 객체를 동기화 객체 안으로 캡슐화 한 것이다.
      - 비공개 락 객체 관용구는 무조건적 스레드 안전 클래스에서만 사용 할 수 있다.
        조건부 스레드 안전 클래스에서는 특정 호출 순서에 필요한 락이 무엇인지를 클라이언트에게 알려줘야 하므로 이 관용구를 사용할 수 없다.

> 앞의 코드에서 lock 필드를 `final` 로 선언했다. 이는 우연히라도 락 객체가 교체되는 일을 예방해준다. 락이 교체되면 끔찍한 결과로 이어진다. (아이템 78) 변경 가능성을 최소화 (아이템 17)를 따라 다시 한번 락 필드의 변경 가능성을 최소화한 것이다. 이처럼 **락 필드는 항상 `final` 로 선언하라. (**일반적인 락이든, `java.util.concurrent.locks` 에서 가져온 락이든**)**

- 정리
  - 모든 클래스가 자신의 스레드 안전성 정보를 명확히 문서화해야 한다.
    - 정확한 언어로 명확히 설명하거나 스레드 안전성 애너테이션(`@Immutable`, `@ThreadSafe` (무조건 스레드 안전, 조건부 스레드 안전), `@NotThreadSafe`)을 사용할 수 있다.
  - `synchronized` 한정자는 문서화와 관련없다.
  - 조건부 스레드 안전 클래스는 메서드를 어떤 순서로 호출할 때 외부 동기화가 요구되고, 그때 어떤 락을 얻어야 하는지도 알려줘야 한다.
  - 무조건적 스레드 안전 클래스를 작성할 때는 `synchronized` 메서드가 아닌 비공개 락 객체를 사용하자. 이렇게 해야 클라이언트나 하위 클래스에서 동기화 매커니즘을 깨뜨리는 걸 예방 할 수 있다.

▼ 83. 지연 초기화는 신중히 사용하

- 지연 초기화 (lazy initaliztion) - (cp 1 싱글톤 관련 내용시 했던 것)
  - 필드의 초기화 시점을 그 값이 처음 필요할때까지 늦추는 기법이다.
  - 값이 전혀 쓰이지 않으면 초기화도 결코 일어나지 않는다.
  - 이 기법은 정적 필드와 인스턴스 필드 모두에 사용할 수 있다.
  - 지연 초기화는 주로 최적화 용도로 사용되지만, 클래스와 인스턴 초기화 때 발생하는 위험한 순환 문제를 해결하는 효과도 있다.

- 지연 초기화는 양날의 검이다.
  - 필요할 때까지 하지 말라.
  - 클래스 혹은 인스턴스 생성 시의 초기화 비용은 줄지만
    그 대신 지연 초기화하는 필드에 접근하는 비용이 커진다.
  - 지연 초기화하려는 필드들 중 결국 초기화가 이뤄지는 비율에 따라,
    실제 초기화에 드는 비용에 따라,
    초기화된 각 필드를 얼마나 빈번히 호출하느냐에 따라,
    지연 초기화가 실제로는 성능을 느려지게 할 수도 있다.
- 지연 초기화가 필요한 시점
  - 해당 클래스의 인스턴스 중 그 필드를 사용하는 인스턴스의 비율이 낮은 반면, 그 필드를 초기화하는 비용이 크다면 지연 초기화가 제 역할을 해줄 것이다.
    쉽게 말해, 사용은 자주 하지 않지만, 초기화 비용이 아주 크다면 사용해볼만 하다.
  - 하지만, 이걸 체크 하는 방법으로는 지연 초기화 적용 전후의 성능을 측정 해야한다.
- 멀티스레드 환경에서 지연 초기화
  - 멀티스레드 환경에서는 지연 초기화를 하기가 까다롭다.
  - 지연 초기화하는 필드를 둘 이상의 스레드가 공유한다면 어떤 형태로든 반드시 동기화해야 한다. 그렇지 않으면 심각한 버그로 이어진다. (아이템 78)
- **대부분의 상황에서는 일반적인 초기화가 지연 초기화보다 좋다.**
  - ex) 인스턴스 필드를 초기화하는 방법 - `final` 한정자를 사용함 (아이템 17)

```
private final FieldType field = computeFieldValue();
```

- 지연 초기화가 초기화 순환성을 깨뜨릴 것 같으면 `synchronized` 단 접근자를 사용하자.
  - 이 방법이 가장 간단하고 명확한 대안이다.
  - ex) 인스턴스 필드의 지연 초기화 - `synchronized` 접근자 방식

```
private FieldType field;

private synchronized FieldType getField() {
    if (field == null)
        field = computeFieldValue();
    return field;
}
```

  - 위 두 관용구 (보통의 초기화, `synchronized` 접근자를 사용한 지연 초기화)는 정적 필드에도 똑같이 적용된다. 물론 필드와 접근자 메서드에 `static` 한정자 추가
- **성능 때문에 정적 필드를 지연 초기화한다면 지연 초기화 홀더 클래스 관용구를 사용하자**
  - 클래스가 처음 쓰일 때 비로소 초기화 된다는 특성을 이용한 관용구이다.
  - ex) 정적 필드용 지연 초기화 홀더 클래스 관용구

```
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
}

private static FieldType getField() { return FieldHolder.field; }
```

    - `getField` 가 처음 호출되는 순간 `FieldHolder.field` 가 처음 읽히면서, 비로소 `FieldHolder` 클래스 초기화를 촉발한다.
    - 이 관용구의 멋진 점은 `getField` 메서드가 필드에 접근하면서 동기화를 전혀 하지 않으니 성능이 느려질 거리가 전혀 없다는 것이다.

- **성능 때문에 인스턴스 필드를 지연 초기화한다면 이중검사 관용구를 사용하자**
  - 초기화된 필드에 접근할 때의 동기화 비용을 없애준다. (아이템 79)
  - 필드 값을 두 번 검사하는 방식으로, 한 번은 동기화 없이 검사하고, (필드가 아직 초기화가 되지 않았다면) 두 번째는 동기화하여 검사한다.
    두 번째 검사에서도 필드가 초기화되지 않았을 때만 필드를 초기화한다.
  - 필드가 초기화 된 후로는 동기화하지 않으므로 해당 필드는 반드시 `volatile` 로 선언해야한다.
  - ex) 인스턴스 필드 지연 초기화용 이중검사 관용구

```
private volatile FieldType field;

private FieldType getField() {
    FieldType result = field;
    if (rusult != null) { // 첫 번째 검사
        return result;

    synchronized (this) {
        if (field == null) // 두 번째 검사 - 락 사용
            field = computeFieldValue();
        return field;
    }
}
```

    - `result` 변수는 필드가 이미 초기화된 상황에서 그 필드를 딱 한 번만 읽도록 보장하는 역할을 한다. 반드시 필요하지는 않지만 성능을 높여주고, 저수준 동시성 프로그래밍에 표준적으로 적용되는 더 우아한 방법이다.

- 이중검사에는 언급해둘 만한 두 가지가 있다.

1. 이따금 반복해서 초기화해도 상관없는 인스턴스 필드를 지연 초기화해야 할 때가 있는데, 이런 경우라면 이중 검사에서 두 번째 검사를 생략할 수 있다. (단일 검사 관용구)

  - ex) 단일 검사 관용구

```
private volatile FieldType field;

private FieldType getField() {
    FieldType result = field;
    if (result == null)
        field = result = computeFieldValue();
    return result;
}
```

2. 모든 스레드가 필드 값을 다시 계산해도 상관없고 필드의 타입이 `long` 과 `double` 을 제외한 다른 기본 타입이라면, 단일 검사의 필드 선언에서 `volatile` 한정자를 없애도된다.
   이 변종은 짜릿한 단일 검사(racy single-check) 관용구라고 부른다.
   - 어떤 환경에서는 필드 접근 속도를 높여주지만, 초기화가 스레드당 최대 한 번 더 이뤄질 수 있다. 보통은 거의 쓰지 않는다.

- 정리
  - 모든 지연 초기화 기법(홀더 클래스, 더블 체크, 싱글 체크)은 기본 타입 필드와 객체 참조 필드 모두에 적용할 수 있다.
    - 이중검사와 단일검사 관용구를 수치 기본 타입 필드에 적용한다면 필드 값을 `null` 대신 0과 비교하면 된다.
  - 대부분의 필드는 지연시키지 말고 곧바로 초기화해야한다.
  - 성능 때문에 혹은 위험한 초기화 순환을 막기 위해 꼭 지연 초기화를 써야 한다면 올바른 지연 초기화 기법을 사용하자.
  - **인스턴스 필드**에는 **이중검사 관용구**를 **정적 필드**에는 **지연 초기화 홀더 클래스 관용구**를 사용하자. **반복해 초기화해도 괜찮은 인스턴스 필드**에는 **단일검사 관용구**도 고려 대상이다.

▼ 84. 프로그램의 동작을 스레드 스케줄러에 기대지 말라

- 개요
  - 여러 스레드가 실행 중이면 운영체제의 스레드 스케줄러가 어떤스레드를 얼마나 오래 실행할지 정한다.
  - 정상적인 운영체제라면 이 작업을 공정하게 수행하지만 구체적인 스케줄링 정책은 운영체제마다 다를 수 있다. 따라서 잘 작성된 프로그램이라면 이 정책에 좌지우지돼서는 안된다.
  - **정확성이나 성능이 스레드 스케줄러에 따라 달라지는 프로그램이라면 다른 플랫폼에 이식하기 어렵다.**
- 견고하고 빠릿하고 이식성 좋은 프로그램을 작성하는 방법
  1. 실행 가능한 스레드의 평균적인 수를 프로세서 수보다 지나치게 많아지지 않도록 한다.
     - 스레드 스케줄러가 고민할 거리가 줄어드는 방법이다.
     - 실행 준비가 된 스레드들은 맡은 작업을 완료할 때까지 계속 실행되도록 만들자.
     - 실행 가능한 스레드 수를 적게 유지 하는 기법
       - 무언가 유용한 작업을 완료한 후에는 다음 일거리가 생길 때까지 대기하도록 하는 것이다.
       - **스레드는 당장 처리해야 할 작업이 없다면 실행돼서는 안 된다.**
       - 실행자 프레임 워크를 예로 들면, 스레드 풀 크기를 적절히 설정하고 작업은 짧게 유지하면 된다. 단, 너무 짧으면 작업을 분배하는 부담이 오히려 성능을 떨어뜨릴 수 있다.
  2. 스레드는 절대 바쁜 대기(busy waiting) 상태가 되면 안된다.
     - 공유 객체의 상태가 바뀔 때까지 쉬지 않고 검사해서는 안 된다는 뜻이다.
     - 바쁜 대기는 스레드 스케줄러의 변덕에 취약할 뿐 아니라, 프로세서에 큰 부담을 주어 다른 유용한 작업이 실행될 기회를 박탈한다.
     - ex) 삐딱한 `CountDownLatch` 구현 (바쁜 대기 버전)

       ```
       public class SlowCountDownLatch {
           private int count;

           public SlowCountDownLatch (int count) {
               if (count < 0)
                   throw new IllegalArgumentException(count + " < 0");
               this.count = count;
           }

           public void await() {
               while (true) {
                   synchronized (this) {
                       if (count == 0)
                           return;
                   }
               }
           }

           public synchronized void countDown() {
               if (count != 0)
                   count--;
           }
       }
       ```

       - 이런 시스템은 성능과 이식성이 떨어질 수 있다.
  3. `Thread.yield` 를 써서 문제를 고치려고 하지 말자.
     - 특정 스레드가 다른 스레드들과 비교해 CPU 시간을 충분히 얻지 못해도 사용해선 안된다. 증상이 어느 정도 호전될 수 있지만, 이식성은 그렇지 않다.
     - `Thread.yield` 는 테스트할 수단도 없다. 차라리 1 번 방법을 조치해보자..
  4. 스레드 우선순위를 조절 하려고 하지말자.
     - 스레드 우선순위는 자바에서 이식성이 가장 나쁜 특성에 속한다.
     - 스레드 몇 개의 우선순위를 조율해서 애플리케이션의 반응 속도를 높일 수 있겠으나, 그렇게 해야할 상황도 드믈고 이식성도 떨어진다.

        - 어차피 스레드 우선순위를 변경해도 근본적인 문제가 아니므로, 조절하지 말자.
- 정리
    - 프로그램의 동작을 스레드 스케줄러에 기대지 말자.
        - 견고성과 이식성을 모두 해치는 행위이다.
    - 같은 이유로 Thread.yield 과 스레드 우선순위에 의존해서는 안 된다.
        - 이 기능들은 스레드 스케줄랑 제공하는 힌트 일 뿐이다.
        - 스레드 우선순위는 이미 잘 동작하는 프로그램의 서비프 품질을 높이기 위해 드물게 사용될 수 있으나, 간신히 동작하는 프로그램을 '고치는 용도'로 사용하면 안 된다.