**용어 정리**

-임계영역(Critical Section) : 둘 이상의 스레드가 공유데이터에 동시에 접근해서 문제가 될 수 있는 코드영역

-동기화 : 스레드간 동시 접근 가능한 공유영역에 대해서 접근제어 및 접근 순서 컨트롤 해주는 작업

1) 상호배제 (Mutual Exclusion) : 둘 이상의 스레드가 임계영역에 동시에 접근하는 것을 막는 것

2) 협력 (Cooperation) : 접근 순서 컨트롤

-일반적인 동기화 처리 방식(OS) : 뮤텍스(Mutex), Semaphore(세마포어)

**-자바는?**

-동기화 코드 영역(synchronized)에 진입하기 위해 모니터락을 획득한다.



-EntrySet : 모니터락을 기다리는 스레드를 담는 자료구조

-WaitSet : 모니터가 notify 해줄 때까지 기다리는 스레드를 담는 자료구조



**스레드 상태**

package java.lang;  
public class Object {  
public final native void notify();  
public final native void notifyAll();  
public final native void wait(long timeout) throws InterruptedException;  
public final void wait(long timeout, int nanos) throws InterruptedException {  
 if (timeout < 0) {  
 throw new IllegalArgumentException("timeout value is negative");  
 }  
  
 if (nanos < 0 || nanos > 999999) {  
 throw new IllegalArgumentException(  
 "nanosecond timeout value out of range");  
 }  
  
 if (nanos >= 500000 || (nanos != 0 && timeout == 0)) {  
 timeout++;  
 }  
  
 wait(timeout);  
 }  
public final void wait() throws InterruptedException {  
 wait(0);  
 }  
}

**자바에서 동기화**

1) 상호배제 (Mutual Exclusion) : synchronized (keyword)

2) 협력 (Cooperation) : wait, notify, notifyAll (object)

**락 공정성**

synchronized 블록은 쓰레드 진입 순서에 대한 어떤 보장도 해주지 않음.

wait / notify도 동일.

🡺ReentrantLock  (명시적 lock)으로 unfair, fair 선택할 수 있음

<https://parkcheolu.tistory.com/21>

**ITEM 78 공유 중인 가변 데이터는 동기화해 사용하라**

**synchronized**

1) 배타적 실행 (Mutual Exclusion) : 하나의 스레드만 접근 (객체의 일관된 상태)

2) 가시성 (Visibility) 문제 (한 스레드에서 변경한 특정 메모리 값을 다른 스레드에서 제대로 읽지 못하는 경우) 해결

**가시성 문제 원인**

**1) Java Memory Model**



**2) reoreder**

reoreder란 최적화를 위해 컴파일이나 Java 가상머신이 프로그램의 처리순서를 바꾸는 것을 말한다.

**volatile**

1) 배타적 실행 (X)

2) 가시성 문제 해결 (O)



🡺하나의 Thread에서 write 하고 나머지 Thread에서 read 할 때 적합

long stat = 324L;  
volatile int val = stat+10;

🡺Thread1은 324를 가지고 갈 수도 있고, Thread2는 334를 가지고 갈 수도 있다. (안전 실패)

🡺이럴 때는 synchronized를 사용해서 작업 자체를 원자화 시킨다.

**Atomic (넌 블로킹)**

1) 배타적 실행 (O) / lock 아님

2) 가시성 문제 해결 (O)

락 대신 저수준의 하드웨어에서 제공하는 CAS(compare-and-swap) 방식 사용

대상 메모리, 기존 값, 새로 설정할 값

public final long getAndAddLong(Object o, long offset, long delta) {  
 long v;  
 do {  
 v = getLongVolatile(o, offset);  
 } while (!weakCompareAndSetLong(o, offset, v, v + delta));  
 return v;  
}

**가장 좋은 방법?**

불변 데이터만 공유 하거나, 가변 데이터는 단일 스레드에서만 사용하자

가변 데이터를 공유 할꺼면 반드시 동기화를 해야 한다.

**ITEM79 과도한 동기화는 피하라**

과도한 동기화는 성능을 떨어뜨리고, 교착상태에 빠뜨리고, 예측할 수 없는 동작을 발생하기도 한다.

**외계인 메서드**

-동기화된 영역 안에서 재정의할 수 있는 메서드

-클라이언트가 넘겨준 함수 객체

🡺무슨 일을 할지 알지 못하며 통제할 수도 없다.

**예제1)**

public class ObservableSet<E> extends ForwardingSet<E> {  
 private final List<SetObserver<E>> observers = new ArrayList<>();  
  
  
 */\*\*  
 \* 외계인 메소드  
 \*/* 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;  
 }

public static void main(String[] args) {  
 ObservableSet<Integer> set = new ObservableSet<>(new HashSet<>());  
 // 옵저버 등록  
 set.addObserver((s, e) -> System.*out*.println(e));  
  
 for (int i = 0; i < 100; i++) {  
 set.add(i);

}  
 }  
}

🡺크게 이슈 없음

**값이 23일 때, 자기 자신을 제거 하는 관찰자를 추가하면??**

public static void main(String[] args) {  
 ObservableSet<Integer> set = new ObservableSet<>(new HashSet<>());  
  
 // 순회도중에 remove가 발생하면 ConcurrentModificationException 에러가 발생한다.  
 // 콜백은 동시성을 깨뜨릴 수 있다.  
 set.addObserver(new SetObserver<>() {  
 public void added(ObservableSet<Integer> s, Integer e) {  
 System.*out*.println(e);  
 if (e == 23) {  
 s.removeObserver(this);  
 }  
 }  
 });  
  
 for (int i = 0; i < 100; i++) {  
 set.add(i);  
 }  
}

🡺ConcurrentModificationException 에러남

****

removeObserver와 notifyElementAdded가 synchronized 하기 때문에 동기화 될 것이라고 기대했지만, 자기 자신이 콜백을 거쳐 되돌아와 수정하는 것까지 보장하지 못한다.

**결론 : 자기 자신이 콜백을 통해 수행되는 건 동시성으로 막을 수 없다.**

**예제2)**

**별도의 스레드로 제거를 하면?**

public static void main(String[] args) {  
 ObservableSet<Integer> set = new ObservableSet<>(new HashSet<>());  
  
 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();  
 }  
 }  
 }  
 });  
  
 for (int i = 0; i < 100; i++) {  
 set.add(i);  
 }  
}

🡺교착상태 발생

1) 메인 스레드에서 순회하면서 서브 스레드에서 관찰자 제거 하기를 기다림

2) 서브 스레드는 메인 스레드의 락이 끝나길 기다림

**해결1)**

**앞의 예외나 교착상태를 막는 방법 (복사해서 사용)**

// 외계인 메서드를 동기화 블록 바깥으로 옮겼다.  
// 관찰자 리스트를 복사해 쓰면 락 없이도 안전하게 순회할 수 있다.  
// 예외와 교착이 사라진다. (open call)  
private void notifyElementAdded(E element) {  
 List<SetObserver<E>> snapshot = null;  
 synchronized (observers) {  
 snapshot = new ArrayList<>(observers);  
 }  
 for (SetObserver<E> observer : snapshot) {  
 observer.added(this, element);  
 }  
}

**해결2) CopyOnWriteArrayList**

private final List<SetObserser<E>> observers = new CopyOnWriteArrayList<>();  
  
public void addObserver(SetObserver<E> observer) {  
 observers.add(observer);  
}  
  
public boolean removeObserver(SetObserver<E> observer) {  
 return observers.remove(observer);  
}  
  
public void notifyElementAdded(E element) {  
 for (SetObserver<E> observer : observers) {  
 observers.added(this, element);  
 }  
}

**CopyOnWriteArrayList.class**

public void add(int index, E element) {  
 synchronized (lock) {  
 Object[] elements = getArray();  
 int len = elements.length;  
 if (index > len || index < 0)  
 throw new IndexOutOfBoundsException(*outOfBounds*(index, len));  
 Object[] newElements;  
 int numMoved = len - index;  
 if (numMoved == 0)  
 newElements = Arrays.*copyOf*(elements, len + 1);  
 else {  
 newElements = new Object[len + 1];  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 System.*arraycopy*(elements, index, newElements, index + 1,  
 numMoved);  
 }  
 newElements[index] = element;  
 setArray(newElements);  
 }  
}

public E remove(int index) {  
 synchronized (lock) {  
 Object[] elements = getArray();  
 int len = elements.length;  
 E oldValue = *elementAt*(elements, index);  
 int numMoved = len - index - 1;  
 if (numMoved == 0)  
 setArray(Arrays.*copyOf*(elements, len - 1));  
 else {  
 Object[] newElements = new Object[len - 1];  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 System.*arraycopy*(elements, index + 1, newElements, index,  
 numMoved);  
 setArray(newElements);  
 }  
 return oldValue;  
 }  
}

public E get(int index) {  
 return *elementAt*(getArray(), index);  
}

static <E> E elementAt(Object[] a, int index) {  
 return (E) a[index];  
}

🡺수정할 일이 드물고 순회만 빈번할 때 최적화

**동기화**

동기화 영역에서는 가능한 한 일을 적게 하자. (과도한 동기화를 피하자)

락을 얻는데 드는 시간보다 병렬로 실행할 기회를 잃고, 모든 코어가 메모리를 일관되게 보기 위한 지연시간이 진짜 비용이다.

**가변 클래스를 작성하려면?**

1) 동기화를 전혀 하지 말고 가변 클래스를 동시에 사용해야 하는 클래스가 외부에서 동기화

ex) java.util

2) 동기화를 내부에서 수행해 스레드 안전한 클래스로 만들자

🡺클라이언트가 외부에서 객체 전체에 락을 거는 것보다 동시성을 월등히 개선할 수 있을 경우만

ex) java.util.concurrent

Vector, HashTable, StringBuffer, Random 등 내부적으로 동기화 진행했지만 성능은 별로 좋지 않다.

**좀 더 자세한 내용은 자바병렬 프로그래밍(에이콘) 참고**

**ITEM80 스레드보다는 실행자, 태스크, 스트림을 애용하라**

**1) 순차적으로 처리 (성능 떨어짐)**

**2) 요청이 올 때마다 새로운 스레드를 생성 (과거)**

public class ThreadPerTaskWebServer {  
 public static void main(String[] args) {  
 while (true) {  
 Runnable task = () -> {  
 System.*out*.println("test");  
 };  
 new Thread(task).start();  
 }  
 }  
}

**장점 :** 동시에 여러 작업 병렬 처리 가능, 순차적인 방법에 비해서 속도가 크게 향상

**단점 :** 스레드 라이프 사이클 문제(스레드 매번 생성), 자원 낭비(불필요한 스레드 생성(idle)), 안정성(OOM)

🡺자원 관리 측면에서 허점이 있다.

**3) JDK5에서 실행자 프레임워크(Executor) 추가**

**Executor 인터페이스 다이어그램**



**실행자 프레임워크(Executor)**

**-**작업 단위(Task) + 실행 메커니즘(ExecutorService)

**실행자 서비스(ExecutorService)**

-Executor의 라이프 사이클 관리

-작업 단위(Callable) 확장



**스레드풀 Factory (Executors)**

-newFixedThreadPool(n)

-newCachedThreadPool

-newSingelThreadExecutor

-newScheduledThreadPool

-newWorkStealingPool (JDK8)

**ForkJoinPool**

-병렬 스트림

-Executors.newWorkStealingPool

-Completablefuture.Async

**ThreadPool vs ForkJoinPool**

**ThreadPool**



**ForkJoinPool**



<https://javacan.tistory.com/entry/134>

<https://hamait.tistory.com/612>

**ITEM81 wait와 notify보다는 동시성 유틸리티를 애용하라**

wait와 notify는 올바르게 사용하기가 아주 까다로우니 고수준 동시성 유틸리티를 사용

**Java.util.concurrent의 고수준 유틸리티는 세 범주로 나눌 수 있다.**

~~1) 실행자 프레임워크~~

2) 동시성 컬렉션(concurrent collection)

3) 동기화 장치(synchronizer)

**1) 동시성 컬렉션**

-List, Queue, Map 같은 표준 컬렉을 동시성을 가미해 구현한 고성능 컬렉션

-높은 동시성에 도달하기 위해 각자 내부에서 동기화

-여러 기본 동작을 하나의 원자적 동작으로 묶는 ‘**상태 의존적 수정**’ 메서드

ex) ConcurrentMap의 putIfAbsent(key, value)

**HashTable < Collections.synchronizedMap < ConcurrentHashMap**



**문제점**

-HashMap : Thread Safe 하지 않다.

-Hashtable, SynchronizedMap : Thread Safe 하나, 두개 이상의 연산 필요할 때, 외부 동기화 처리

**ConcurrentHashMap**

Map을 16개(기본값)으로 나누고 각각(reentrantLock) 병렬적으로 수행

**Synchronized(암묵적) vs ReentrantLock(명시적)**

ReentrantLock은 세부적으로 동기화 가능

private static ReentrantLock *lock* = new ReentrantLock();  
  
public static void main(String[] args) {  
 *lock*.lock();  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 } finally {  
 *lock*.unlock();  
 }  
}

**2) 동기화 장치**

스레드끼리 작업 조율

**BlockingQueue**

BlockingQueue queue = new ArrayBlockingQueue<>(5);  
  
Thread producer = new Thread(new Producer(queue));  
Thread consumer = new Thread(new Consumer(queue));  
  
producer.start();  
consumer.start();

-Queue가 꽉 찼을 때 put block, Queue가 비었을 때 take 시도 block (ReentrantLock)

-Producer-Consumer패턴 적합



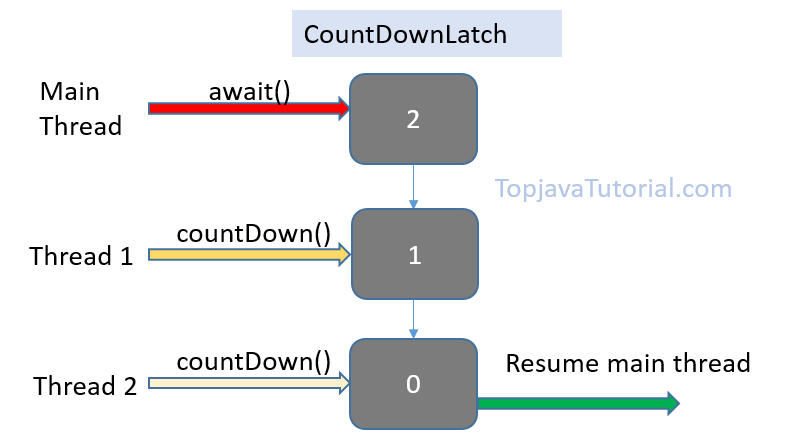
**CountDownLatch**

private static final int *MAX* = 10;  
public static void countdownlatch() throws Exception {  
 CountDownLatch latch = new CountDownLatch(*MAX*);

for (long i = 0; i < *MAX*; i++) {  
 new Thread(new Worker(latch)).start();  
 }  
  
 latch.await(); // MAX count가 다 끝날때까지 기다린다.  
 //latch.await(2000, TimeUnit.MILLISECONDS); // timeout 이후에 종료  
}  
  
private static class Worker implements Runnable {  
 private CountDownLatch latch;  
  
 public Worker(CountDownLatch latch) {  
 this.latch = latch;  
 }  
  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException ex) {  
 ex.printStackTrace();  
 } finally {  
 if (this.latch == null)  
 return;  
  
 latch.countDown();  
 }  
 }  
}

- 어떠한 조작이 지정 횟수만큼 이뤄질 때까지 스레드를 대기시키는 동기화 장치 (메인 스레드에서 취함)

- CountDwonLatch의 개수와 스레드의 수가 다르면 스레드 기아 교착상태 가 발생한다.

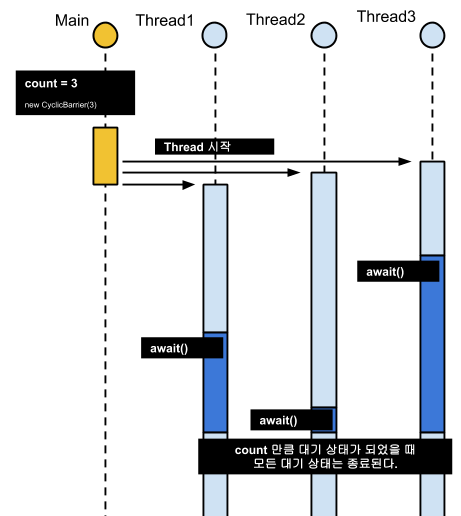


**CyclicBarrier**

private static class Task implements Runnable {  
 private CyclicBarrier barrier;  
 public Task(CyclicBarrier barrier) {  
 this.barrier = barrier;  
 }  
  
 @Override  
 public void run() {  
 try {  
 barrier.await();  
 } catch (InterruptedException ex) {  
 } catch (BrokenBarrierException ex) {  
 }  
 }  
}  
  
public static void main(String args[]) {  
 final CyclicBarrier cb = new CyclicBarrier(3);

Thread t1 = new Thread(new Task(cb), "Thread 1");  
 Thread t2 = new Thread(new Task(cb), "Thread 2");  
 Thread t3 = new Thread(new Task(cb), "Thread 3");  
  
 t1.start();  
 t2.start();  
 t3.start();  
}

-복수 스레드를 특정 장소(Barrier)에 대기시키는 동기화 장치



**CountDownLatch vs CyclicBarrier (좀더 정리)**

둘 다 큰 작업을 작은 작업으로 분리하고, 분리한 작업의 실행을 완료한 뒤에 흐름을 진행

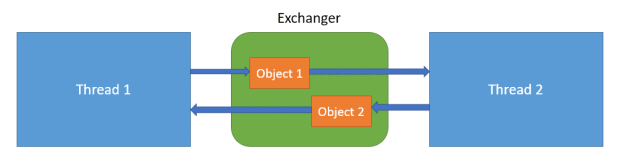
CountDownLatch는 메인 스레드에서 흐름 진행, CyclicBarrier는 개별 스레드에서 ‘다른 스레드’를 기다리기 위한 동기화

CountDownLatch는 단발성, CyclicBarrier는 리셋해서 여러 번 사용가능

**Exchanger (환전상)**

public static void main(String[] args) {  
 Exchanger exchanger = new Exchanger();  
  
 Worker worker1 = new Worker(exchanger, "AAA");  
 Worker worker2 = new Worker(exchanger, "BBB");  
  
 new Thread(worker1).start();  
 new Thread(worker2).start();  
}  
  
private static class Worker extends Thread {  
 Exchanger<String> exchanger;  
 String message;  
  
 Worker(Exchanger<String> exchanger, String message) {  
 this.exchanger = exchanger;  
 this.message = message;  
 }  
  
 public void run() {  
 try {  
 System.*out*.println(this.getName() + " message: " + message);  
  
 // exchange messages  
 message = exchanger.exchange(message);  
  
 System.*out*.println(this.getName() + " message: " + message);  
 } catch (Exception e) {  
 }  
 }  
}

-2개 스레드가 객체를 교환하는 동기화 장치



**Phaser**

public static void main(String[] args) {  
 Phaser phaser = new Phaser();  
  
 Thread t1 = new Worker(phaser, 1000);  
 Thread t2 = new Worker(phaser, 3000);  
 Thread t3 = new Worker(phaser, 10000);  
 t1.start();  
 t2.start();  
 t3.start();  
}  
  
private static class Worker extends Thread {  
 Phaser phaser;  
 int sleep;  
  
 Worker(Phaser phaser, int sleep) {  
 this.phaser = phaser;  
 this.sleep = sleep;  
 }  
  
 public void run() {  
 phaser.register();  
 System.*out*.println(this.getName() + " begin");  
 try {  
 Thread.*sleep*(sleep);  
 } catch (Exception e) {  
 }

phaser.arriveAndAwaitAdvance();  
 System.*out*.println(this.getName() + " middle");  
 try {  
 Thread.*sleep*(sleep);  
 } catch (Exception e) {  
 }  
 System.*out*.println(this.getName() + " end");  
 }  
}

-CyclicBarrier와 유사하지만 좀 더 유연하고 다양한 기능을 제공

**Semaphore**

public class SemaphoreExample {  
 public static void main(String args[]) {  
 Semaphore semaphore = new Semaphore(2);  
 Worker mt1 = new Worker(semaphore);  
 Worker mt2 = new Worker(semaphore);  
 Worker mt3 = new Worker(semaphore);  
 Worker mt4 = new Worker(semaphore);  
 mt1.start();  
 mt2.start();  
 mt3.start();  
 mt4.start();  
 }  
  
 private static class Worker extends Thread {  
 private Semaphore semaphore;  
  
 Worker(Semaphore semaphore) {  
 this.semaphore = semaphore;  
 }  
  
 public void run() {  
 try {  
 semaphore.acquire(); // 리소스 확보  
 try {  
 *sleep*(2000);  
 } catch (Exception e) {  
 }  
 } catch (InterruptedException ie) {  
 } finally {  
 semaphore.release(); // 리소스 반환  
 }  
 }  
 }  
}

-호출할 수 있는 스레드의 수를 제한

-Semaphore(1)은 mutex와 동일

(개인적으로 newFixedThreadPool 랑 비슷)

**wait, notify, notifyAll (스레드의 협력)**

* wait set (스레드 대합실)
  + 스레드는 wait 메소드를 실행하면 동작을 일시 정지하고 wait set에 들어간다.
  + 다음 한 상황이 발생하면 wait set에서 나올 수 있다.

🡺notify, notifyAll, interrupt, wait 메소드 타임아웃

* wait 메소드 (스레드를 wait set에 넣는다)
  + wait 메소드를 사용하면 스레드가 wait set에 들어간다.
  + wait 메소드를 실행하기 위해서 스레드가 락을 가지고 있어야 한다.
  + wait 메서드를 사용할 때는 반드시 반복문 안에서 (허위 각성)

public synchronized void lock() { // 반드시 락을 획득  
 while (busy) { // 조건이 안맞았을 때 wait가 호출되면 스레드를 다시 못깨울 수 있다  
 try {  
 wait(); // 반드시 반복문 안에서

} catch (InterruptedException e) {  
  
 }  
 }  
 busy = true;  
}

* notify 메소드 (스레드를 wait set에서 꺼낸다)
  + notify 메소드를 사용하면 wait set에 있는 스레드 한개를 꺼낸다.
  + notify 메소드를 실행하기 위해서는 wait 메소드와 마찬가지로 스레드가 락을 가지고 있어야 한다. (규칙)
  + notify 메소드를 실행했을 때 wait set에 대기중인 스레드가 여러개 있는 경우 어느 스레드가 선택될지는 정해져 있지 않다.
* notifyAll 메소드 (모든 스레드를 wait set에서 꺼낸다)
  + notifyAll 메소드를 사용하면 wait set에 있는 모든 스레드를 꺼낸다.
  + notify가 한 개의 스레드를 깨우는데 반해 notifyAll은 모든 스레드를 깨운다.
  + notify보다 notifyAll을 사용하는 편이 낫다.
* wait, notify, notifyAll은 Objectc 클래스의 메소드이다.
* 주의점
  + 락을 가지지 못한 스레드가 wait, notify, notifyAll 메소드를 호출하면 IlegalMonitorStateException이 발생된다.

**ITEM82 스레드 안전성 수준을 문서화하라**

**동시에 호출되는 메서드를 문서화 하지 않을 경우**

-클라이언트 프로그램은 동기화를 충분히 하지 못하거나 (부작용 발생)

-지나치게 동기화를 하거나 (성능 이슈, 교착상태, 예측 할 수 없는 동작)

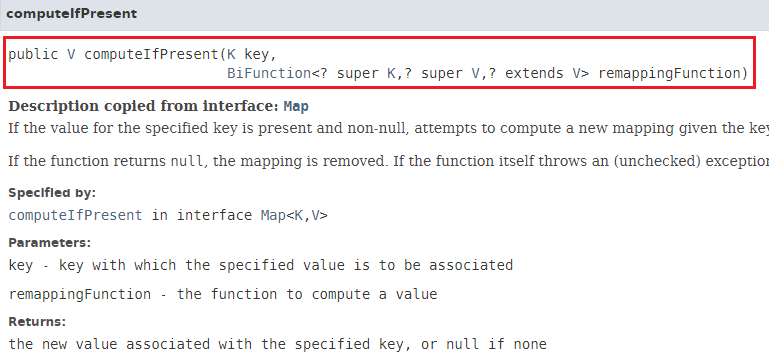
**API 문서에 synchronized 한정자는 포함되지 않는다.**

**Hashtable API**

@Override  
public synchronized V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  
... (생략)

}

**Hashtable Docs**



🡺결국 명확하게 문서에 명시해야 한다.

**스레드 안전성이 높은 순**

1) 불변(immutable) : 동기화 필요 없음 ex)String, Lon, BigInteger

2) 무조건적 스레드 안전 : 내부에서 출실히 동기화 ex) AtomicLong, ConcurrentHashMap

3) 조건부 스레드 안전 : 무조건적 스레드 안전과 같으나 일부 메서드는 외부 동기화 필요

ex) Collections.synchronized 래퍼 메서드

4) 스레드 안전하지 않음 : 외부 동기화 메커니즘 사용 ex) ArrayList, HashMap

5) 스레드 적대적 : 외부 동기화를 사용해도 멀티스레드 환경에서 안전하지 않다. (잘못 설계)

ex)

private static volatile int *nextSerialNumber* = 0;  
  
public static int generateSerialNumber() {  
 return *nextSerialNumber*++;  
}

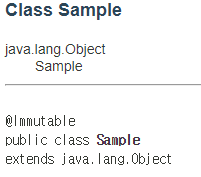
**JCIP(커스텀 주석)**

@Immutable, @ThreadSafe, @NotThreadSafe,

@GuardedBy (반드시 지정된 락을 확보한 상태에서 사용)

ex)

@Immutable  
public class Sample {  
}



**주 의 점**

**조건부 스레드 안전 클래스**

-문서화 시 어떤 순서로 호출할 때 외부 동기화가 필요한지 등 명시

ex) Collections.synchronizedMap

**그 외**

-스레드 안전성은 보통 클래스, 독특한 특성의 메서드라면 메서드에 주석 기재

-열거는 굳이 불변이라고 쓰지 않아도 됨

**-정적 팩터리는 자신이 반환하는 객체의 스레드 안전성을 문서화**

**클래스가 외부에서 사용할 수 있는 락을 제공하면??**

-고성능 동시성 제어 메커니즘(ConcurrentHashMap) 혼용 못함

-클라이언트가 공개락을 쥐고 놓지 않는 서비스 거부 공격을 수행할 수도 있음.

🡺해결 synchronized 메서드 대신 비공개 락 사용 (주의 final 사용)

private final Object lock = new Object(); // 비공개 락  
  
public void foo() {  
 synchronized (lock) { // 보통 this로 사용  
   
 }  
}

**비공개 락**

-무조건 스레드 안전 클래스에서만 사용

-상속용에서도 안전함 (private final)

**ITEM83 지연 초기화는 신중히 사용하라**

지연 초기화는 주로 최적화 용도로 쓰이지만, 클래스와 인스턴스 초기화 때 발생하는 위험한 순환 문제 를 해결하는 효과도 있다.

**지연 초기화 필요할 때**

그 필드를 사용하는 인스턴스의 필드 비율이 낮은 반면, 그 필드를 초기화하는 비용이 클 경우

🡺하지만 성능 측정을 해봐야 된다는거 ㅠㅠ

멀티 스레드에선 지연 초기화하는 공유 필드를 반드시 동기화 해야 한다.

**대부분의 상황에서 일반적인 초기화가 낫다.**

**1) 인스턴스 필드 초기화 일반적인 방법**

private final FieldType field = computeFieldValue();

-final은 필드 변경 불가하다. (불변)

-final은 메모리에 값이 할당 된다. (초기화)

**2) 인스턴스 필드 지연초기화 방법**

private FieldType field2;

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

-**초기화 순환성** 을 깰 것 같으면 synchronized 접근자 이용

**성능 때문에 정적 필드를 지연 초기화 해야 한다면?**

**3) 정적 필드용 지연초기화 방법 (홀더 클래스 관용구)**

private static class FieldHolder {  
 static final FieldType *field* = *computeFieldValue*();  
}  
  
private static FieldType getField() {  
 return FieldHolder.*field*;  
}

-처음 쓰일 때 비로소 초기화 된다는 특성을 이용

-getField 호출 할 때, FileHolder 클래스가 초기화됨. (동기화 X, 성능 UP)

-클래스 초기화할 때 (접근할 때) 만 필드 접근을 동기화, 이후에는 동기화 없이 접근 가능

**성능 때문에 인스턴스 필드를 지연 초기화 해야 한다면?**

**4) 인스턴스 필드 지연초기화 방법 (이중검사 관용구)**

private volatile FieldType field;  
  
private FieldType getField() {  
 FieldType result = field;  
 if (result != null) { // 첫 번째 검사 (락 사용 안 함)  
 return result;  
 }  
  
 synchronized (this) {  
 if (field == null) { // 두 번째 검사 (락 사용)  
 field = computeFieldValue();  
 }  
 return field;  
 }  
}

-초기화된 필드에 접근할 때의 동기화 비용을 없애준다.

-필드가 아직 초기화 되지 않았다면 synchronized

-초기화 이후로 동기화 하지 않으므로 volatile 선언

**5) 인스턴스 필드 지연초기화 방법 (단일검사 관용구)**

private volatile FieldType field;

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

-반복해도 상관 없는 인스턴스 필드를 지연초기화 하는 경우

- field = result = computeFieldValue() 는 내가 생각 할 때 if(result == null) 인 경우 보강 하려고

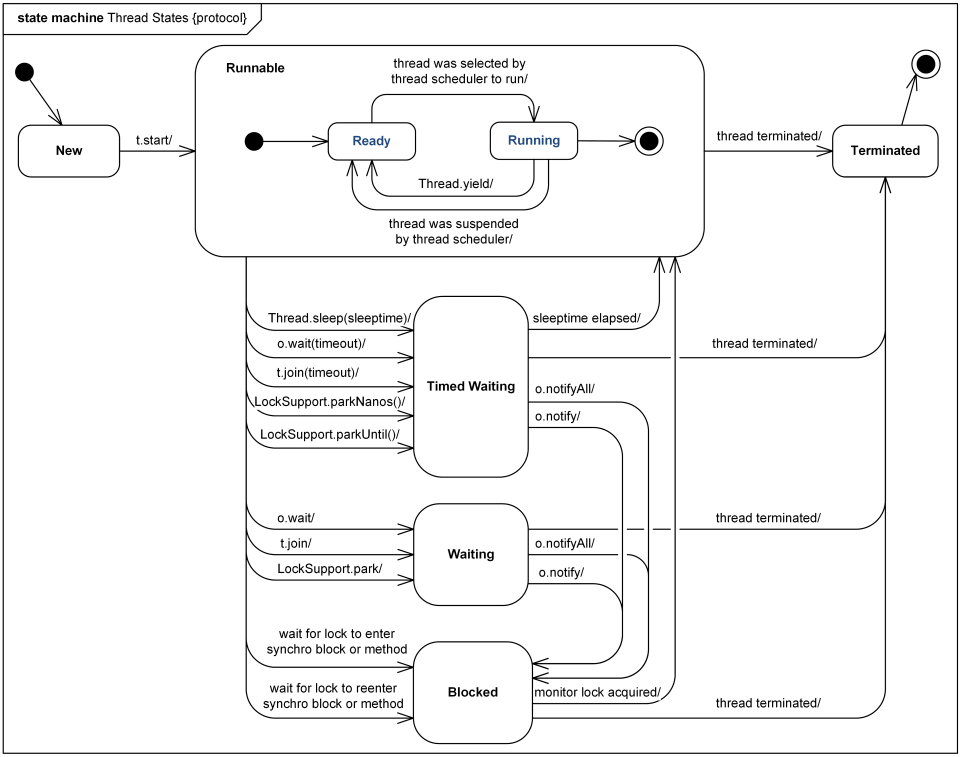
**기본 타입 필드에 적용하려면 null 대신 0과 비교 (long, double 제외 volatile 삭제 가능)**

**정 리**

지연 초기화 보다 일반 초기화 해라.

성능 혹은 초기화 순환을 막기 위해 꼭 지연 초기화를 써야 한다면 올바른 방법으로 써라

**ITEM84 프로그램의 동작을 스레드 스케줄러에 기대지 말라**



**스레드 상태**

**Thread 주요 메서드**

-start() : NEW에서 RUNNABLE로 보낸다. Ready

-run() : RUNNING이 되면 run(), 메서드의 오버라이딩된 로직 수행

-yield() : (우선권이 동일한) 스레드에게 실행 기회를 양보함. Ready -> Running

-join() : 다른 스레드의 종료를 기다린다. wait() / notify()와 흡사

**VM.java**

public enum State { *NEW*,  
 *RUNNABLE*,  
 *BLOCKED*,  
 *WAITING*,  
 *TIMED\_WAITING*,  
 *TERMINATED*;  
}

**스케줄러 하는일 (좀 더 조사)**

-어떤 스레드가 실행되어야 하는지 결정 (Priority)

-얼마나 오랫 동안 실행되어야 하는지 결정

-스레드를 현재 실행중인 상태에서 변경할 때 그 스레드가 어디로 가야되는지 결정

-스레드의 개수가 코어(CPU)의 수보다 많을 경우, 스레드를 어떤 순서에 의해 동시에(context switch) 실행할 것인가 정한다.

플랫폼 마다 스레드 스케줄러 처리하는게 달라지기 때문에(정책) 스레드 스케줄러에 의존하면 안된다.

**실행 가능한 스레드 수를 적게 유지하는 주요 기법**

-작업을 완료한 후에는 다음 작업까지 대기

-스레드는 당장 처리해야 할 작업이 없다면 실행돼서는 안 된다.

ex) 스레드 풀 (스레드 풀 크기를 적절하게 설정하고 작업은 짧게 유지)

<https://javarevisited.blogspot.com/2018/07/java-multi-threading-interview-questions-answers-from-investment-banks.html>

**바쁜 대기(busy waiting)**

공유 객체의 상태가 바뀔 때까지 쉬지 않고 검사

🡺스레드 스케줄러 변덕에 취약, 리소스 부담

Thread thread1 = new Thread(new Subtask1Runnable());  
Thread thread2 = new Thread(new Subtask2Runnable());  
thread1.start();  
thread2.start();  
while( waitinigForThread1() && waitingForThread2() ) {

}

**해결 ???**

Thread thread1 = new Thread(new Subtask1Runnable());  
Thread thread2 = new Thread(new Subtask2Runnable());  
thread1.start();  
thread2.start();  
while( waitinigForThread1() && waitingForThread2() ) {  
 Thread.*sleep*(5000);  
}

**해결 wait, notify, notifyAll 등으로 해결**

public synchronized void put(String cake) throws InterruptedException {  
 System.*out*.println(Thread.*currentThread*().getName() + " puts " + cake);  
 while (count >= buffer.length) { // 조건식  
 wait();  
 }  
 buffer[tail] = cake;  
 tail = (tail + 1) % buffer.length;  
 count++;  
 notifyAll();  
}  
  
public synchronized String take() throws InterruptedException {  
 while (count <= 0) { // 조건식  
 wait();  
 }  
 String cake = buffer[head];  
 head = (head + 1) % buffer.length;  
 count--;  
 notifyAll();  
 System.*out*.println(Thread.*currentThread*().getName() + " take " + cake);  
 return cake;  
}

**Thread.yield**

-(우선권이 동일한) 스레드에게 실행 기회를 양보함. Ready -> Running

🡺yield로 제어해 보려 하지만, JVM 별로 성능이 달라질 수 있다.

**Thread Priority?**

🡺이식성이 가장 나쁘다.

**결론 : 스레드 스케줄러 사용하지 말고(yield, priority) 스레드 개수 최적화 해서 잘 동작하게 하자**