Effective Java

Item 88 : readObject 메서드는 방어적으로 작성하라

Item 89 : readResolve보다는 열거 타입 사용

Item 90 : 직렬화 프록시 사용

Note : Java Thread, Thread Pool

Item 88

readObject 메서드는 방어적으로 작성하라

Quiz

어떤 필드가 직렬화, 역직렬화 될까요?

```
private int age;  2개 사용 위치
private transient int height;  2개 사용 위치
private String name;  2개 사용 위치

@Serial  0개의 사용위치
private void writeObject(ObjectOutputStream out)
    out.defaultWriteObject();
}

@Serial  0개의 사용위치
private void readObject(ObjectInputStream in)
    in.defaultReadObject();
}
```

age, name

```
private int age;  2개 사용 위치
private transient int height;  4개 사용 위치
private String name;  2개 사용 위치

@Serial  0개의 사용위치
private void writeObject(ObjectOutputStream out)
    out.defaultWriteObject();
    out.writeInt(height);
}

@Serial  0개의 사용위치
private void readObject(ObjectInputStream in)
    in.defaultReadObject();
    this.height = in.readInt();
}
```

age, height, name

```
private int age;  2개 사용 위치
private transient int height;  3개 사용 위치
private String name;  2개 사용 위치

@Serial  0개의 사용위치
private void writeObject(ObjectOutputStream out)
    out.defaultWriteObject();
}

@Serial  0개의 사용위치
private void readObject(ObjectInputStream in)
    in.defaultReadObject();
    this.height = in.readInt();
}
```

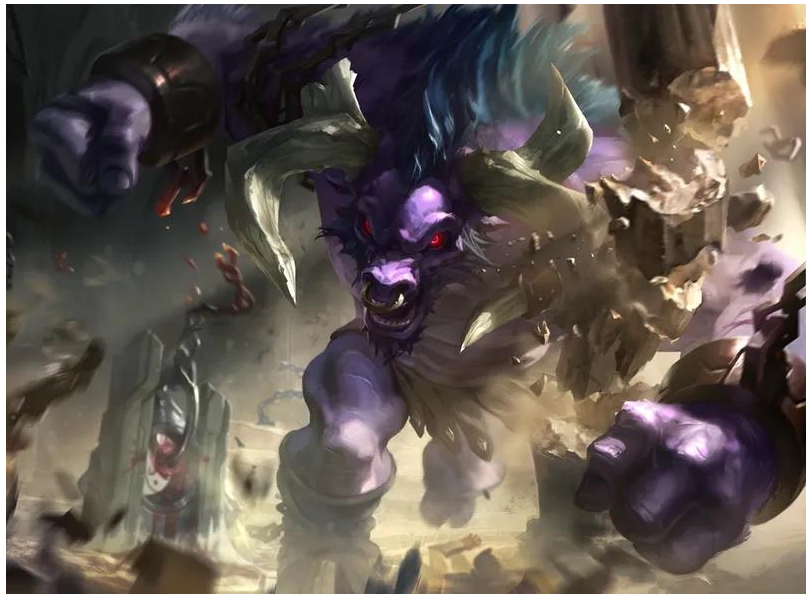
Exception

readObject

V1

```
public final class Period implements Serializable {
    private final Date start;
    private final Date end;

    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());

        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException();
    }
}
```

readObject



```
# 실행 결과, end가 start 보다 과거다. 즉, Period의 불변식이 깨진다.
Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984
```

readObject

https://docs.oracle.com/en/java/javase/11/docs/specs/serialization/protocol.html#grammar-for-the-stream-format

V2

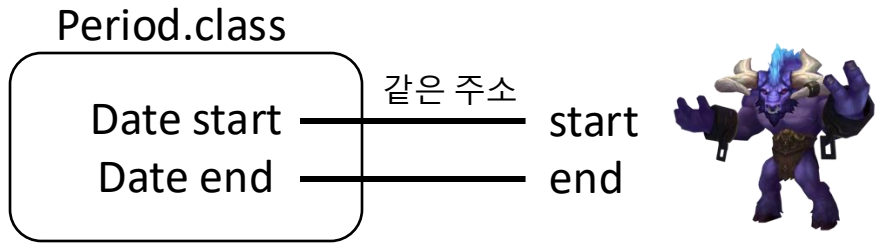
```
public final class Period implements Serializable {
    private final Date start;
    private final Date end;

    private void readObject(ObjectInputStream s) {
        s.defaultReadObject();

        if (start.compareTo(end) > 0) {
            throw new InvalidObjectException();
        }
    }
}
```

공격자 클래스

```
//...
public final Period period;
public final Date start;
public final Date end;

//...
out.writeObject(new Period(new Date(), new Date()));
byte[] ref = { 0x71, 0, 0x7e, 0, 5}
bos.write(ref);
ref[4] = 4;
bos.write(ref);

//...
period = (Period) in.readObejct();
start = (Date) in.readObejct();
end = (end) in.readObejct();
```

write

read



readObject

V3

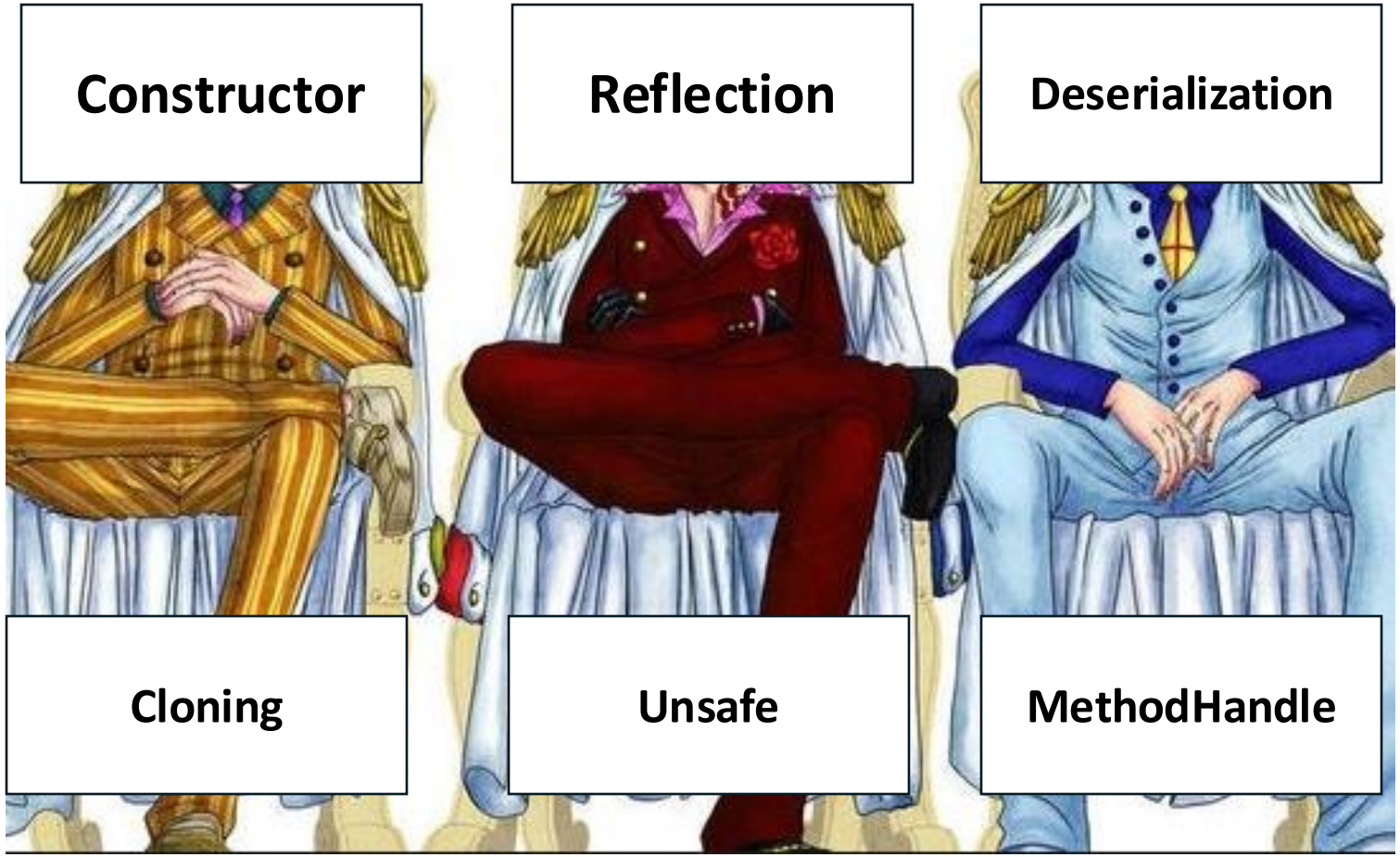
```
public final class Period implements Serializable {
    private Date start;
    private Date end;

    private void readObject(ObjectInputStream s) {
        s.defaultReadObject();

        start = new Date(start.getTime());
        end = new Date(end.getTime());

        if (start.compareTo(end) > 0) {
            throw new InvalidObjectException();
        }
    }
}
```

final 키워드 제거

(transient 필드 제외)
유효성 검사를 하지 않는 생성자를 만들 수 있나?

아니라면 readObject를 재정의하자

인스턴스 생성


Constructor
Reflection
Deserialization
Cloning
Unsafe
MethodHandle

Item 89

인스턴스 수를 통제해야 한다면 열거 타입

Singleton

```
public class Elvis implements Serializable {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() {
    }
}
```

```
Elvis elvis = Elvis.INSTANCE;
out.writeObject(elvis);

// ...
Elvis deserializedElvis = (Elvis) in.readObject();
assertEquals(elvis, deserializedElvis); // Failed
```

Singleton

```
public class Elvis implements Serializable {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() {
    }

    private void readObject(ObjectInputStream s) {
        s.defaultReadObject();
        System.out.println("readObject");
    }

    private Object readResolve() {
        System.out.println("readResolve");
        return INSTANCE;
    }
}
```

호출순서

1

2

새롭게 만든 Elvis 인스턴스는 GC로 정리됨

defaultWriteObject, defaultReadObject 메서드는 static 변수를 처리하지 않는다.

필요하다면 직접 수동으로 처리해야 함

싱글톤의 모든 필드는 transient로 선언해야 한다.

```
public class ElvisImpersonator {
    private static final byte[] serializedForm = {
            (byte) 0xac, (byte) 0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x05,
            0x45, 0x6c, 0x76, 0x69, 0x73, (byte) 0x84, (byte) 0xe6,
            (byte) 0x93, 0x33, (byte) 0xc3, (byte) 0xf4, (byte) 0x8b,
            0x32, 0x02, 0x00, 0x01, 0x4c, 0x00, 0x0d, 0x66, 0x61, 0x76,
            0x6f, 0x72, 0x69, 0x74, 0x65, 0x53, 0x6f, 0x6e, 0x67, 0x73,
            0x74, 0x00, 0x12, 0x4c, 0x6a, 0x61, 0x76, 0x61, 0x2f, 0x6c,
            0x61, 0x6e, 0x67, 0x2f, 0x4f, 0x62, 0x6a, 0x65, 0x63, 0x74,
            0x3b, 0x78, 0x70, 0x73, 0x72, 0x00, 0x0c, 0x45, 0x6c, 0x76,
            0x69, 0x73, 0x53, 0x74, 0x65, 0x61, 0x6c, 0x65, 0x72, 0x00,
            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x01,
            0x4c, 0x00, 0x07, 0x70, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64,
            0x74, 0x00, 0x07, 0x4c, 0x45, 0x6c, 0x76, 0x69, 0x73, 0x3b,
            0x78, 0x70, 0x71, 0x00, 0x7e, 0x00, 0x02
    };

    public static void main(String[] args) {
        Elvis elvis = (Elvis) deserialize(serializedForm);
        Elvis impersonator = ElvisStealer.impersonator;

        elvis.printFavorites();
        impersonator.printFavorites();
    }

    static Object deserialize(byte[] sf) {
        try (ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(sf);
             ObjectInputStream objectInputStream = new ObjectInputStream(byteArrayInputStream)) {

            return objectInputStream.readObject();

        } catch (IOException | ClassNotFoundException e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

싱글톤의 모든 필드는 transient로 선언해야 한다.

1. Elvis 인스턴스 생성

```
public class Elvis implements Serializable {
    public static final Elvis INSTANCE = new Elvis();
    private String[] favoriteSongs = {"Hound Dog",
                                      "Heartbreak Hotel"};

    private Elvis() {
    }

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

2   private void readObject(ObjectInputStream in) {
        in.defaultReadObject();
    }

    private Object readResolve() {
        return INSTANCE;
    } 6. 정상적으로 싱글톤 리턴
}
```

3. ElvisStrealer 인스턴스 생성

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator;
    private Elvis payload;

    private void readObject(ObjectInputStream in) {
        in.defaultReadObject();
    }  4. payload는 Elvis의 인스턴스를 래퍼런스

    private Object readResolve() {
        impersonator = payload;
        return new String[]{"A Fool Such as I"};
    } 5

    private static final long serialVersionUID = 0;
}
```

열거 타입 싱글톤

```
public enum ElvisEnum {

    INSTANCE;
    private String[] favoriteSongs = {"Hound Dog",
                                      "Heartbreak Hotel"};

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

싱글톤 보장
thread safety
Serialization 보장
reflection 안전

Item 90

직렬화 프록시

serialization proxy pattern

```
class Period implements Serializable {

    private final Date start;
    private final Date end;

    public Period(Date start, Date end) {
        this.start = start;
        this.end = end;
    }

    private static class SerializationProxy implements Serializable {
        private static final long serialVersionUID = ...;
        private final Date start;
        private final Date end;

        public SerializationProxy(Period p) {
            this.start = p.start;
            this.end = p.end;
        }

        private Object readResolve() {
            return new Period(start, end);
        }
    }

    private Object writeReplace() {
        return new SerializationProxy(this);
    }

    private void readObject(ObjectInputStream stream) {
        throw new InvalidObjectException("프록시가 필요합니다.");
    }
}
```

생성자는 1개만
바깥 클래스를 매개변수로

역직렬화

직렬화 시 프록시 인스턴스로 변환

역직렬화 방지용(공격 방지)

Note 1

Java Thread

Note 1

Hardware Thread

	인텔 i7-14700K	라이젠 9700X	애플 M3 Pro
코어	P8+E12	8	P6+E6
스레드	16+12 = 28	16	6+6 = 12

SMT(Simultaneous Multi-Threading) : 동시 멀티스레딩

예전에는 4-Way, 8-Way도 사용됐지만, 현재는 2-Way가 가장 널리 사용됨

인텔의 상표명인 '**하이퍼스레딩(HT)**'이 대명사처럼 사용됨

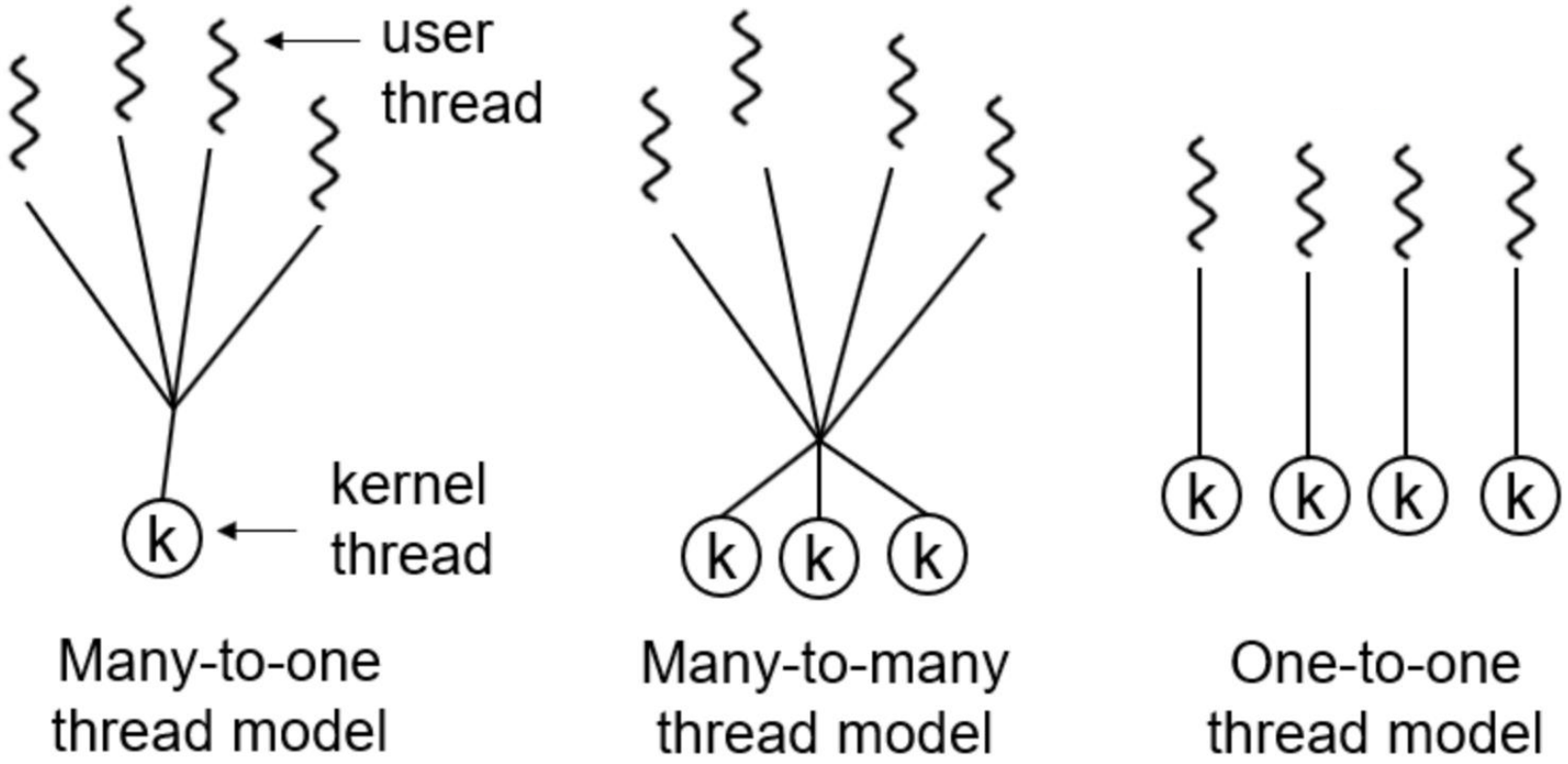
정작 인텔은 HT를 조금씩 빼는..

OS Thread (native, kernel)

- OS 커널 레벨에서 생성되고 관리되는 스레드
- CPU 스케줄링의 단위
- 컨텍스트 스위칭 시에 커널이 개입함
- 생성과 관리 비용이 크다.

User Thread

User Thread가 CPU에서 실행되려면 OS 스레드와 연결되어야 한다.

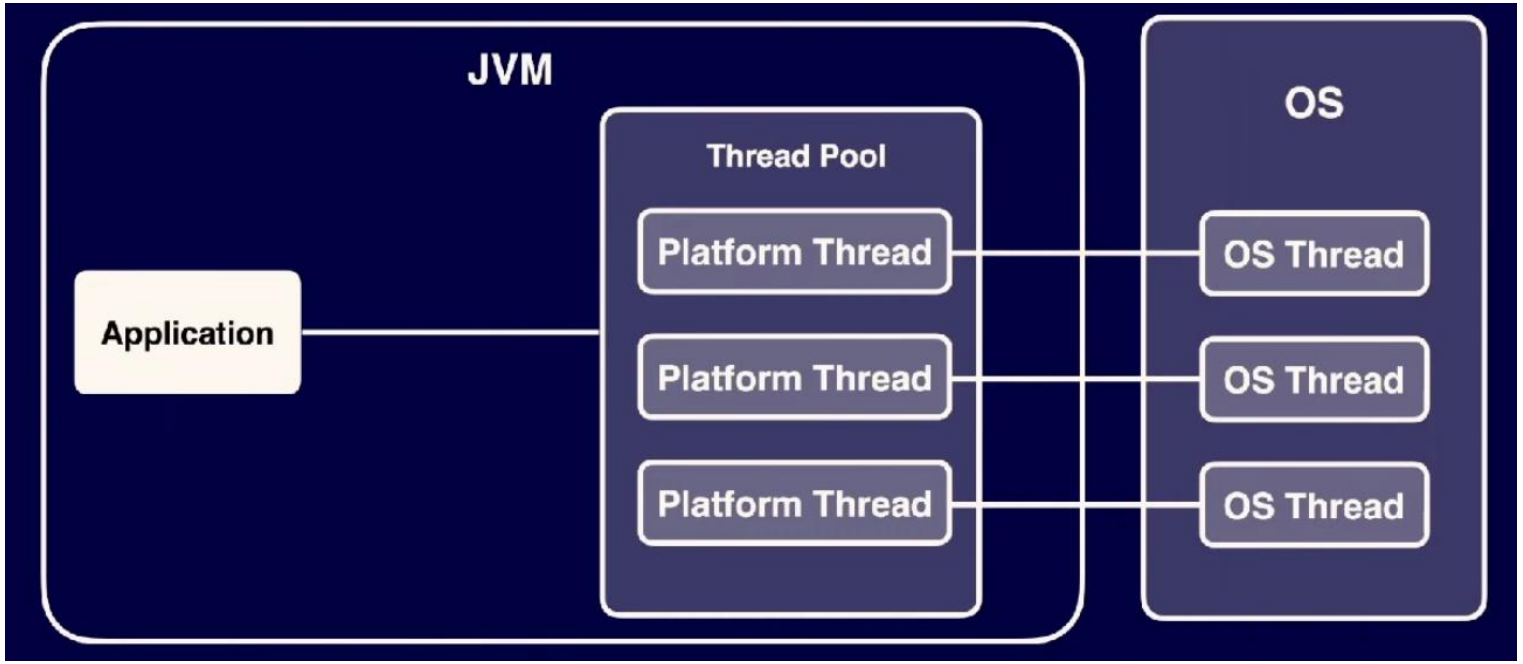


Java Thread

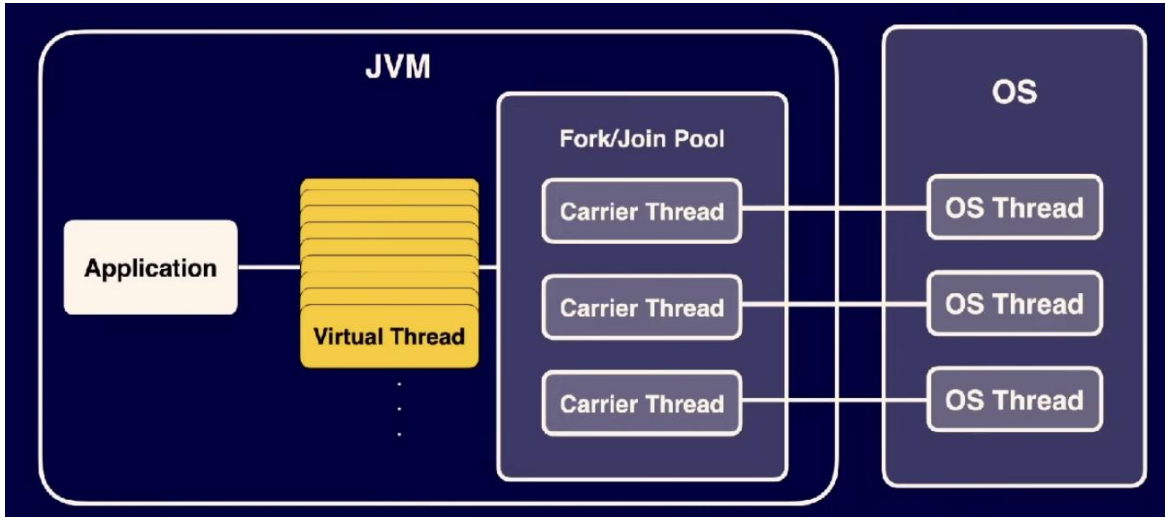
Java의 Thread는 OS Thread를 Wrapping 한 것 (Platform Thread)

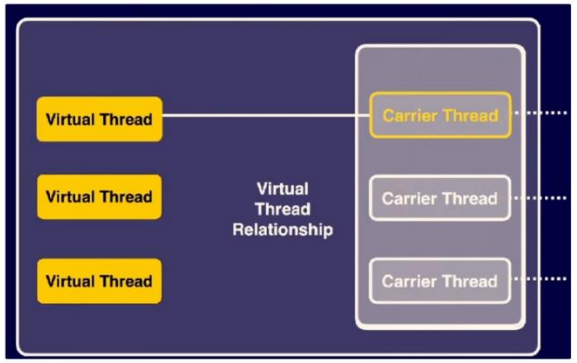
OS Thread는 생성 개수가 제한적이고 생성, 유지하는 비용이 비싸다 -> Thread Pool 사용

```java
// java.lang.Thread.java

private native void start0();

public void start() {
    // ...

    start0();

    // ...
}
```

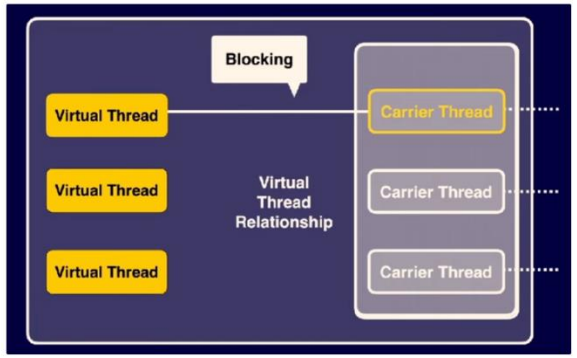


Java Virtual Thread 짧게..

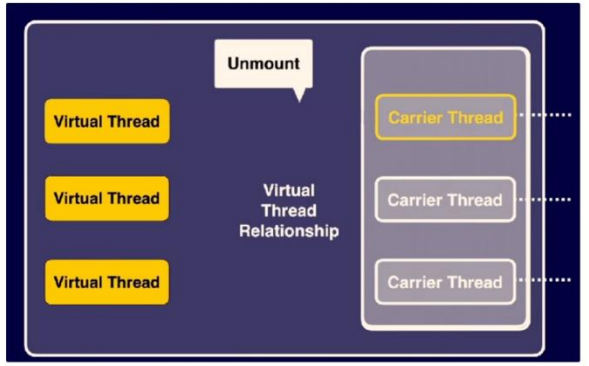




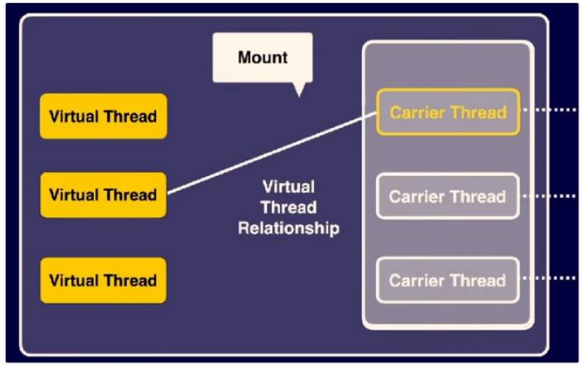
1



2



3



4

Note 1

Java Virtual Thread 짧게..

1. Thread Local 사용시 Heap 메모리 주의
2. synchronized 을 사용하면 Carrier Thread가 block 된다. (pinning)
   - 대신 ReentrantLock을 사용
3. **Overwhelming 주의**
4. I/O Blocking이 발생하는 경우에 적합함. CPU Intensive에는 적합하지 않다.
5. 항상 Daemon 스레드로 동작한다.
6. Platform thread 개수 (https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Thread.html)

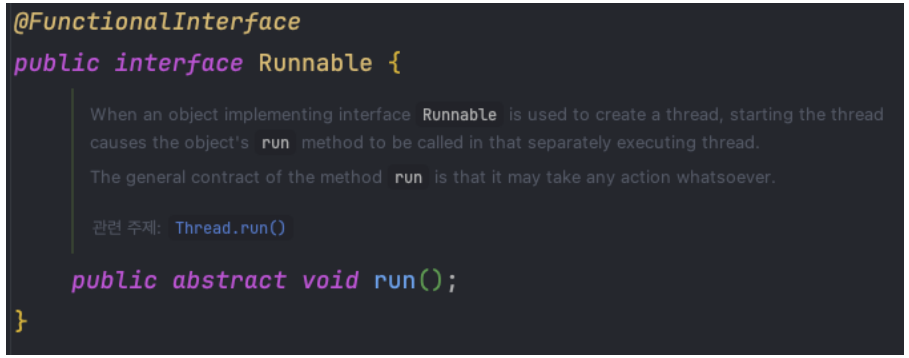
System properties

System property	Description
`jdk.virtualThreadScheduler.parallelism`	The number of platform threads available for scheduling virtual threads. It defaults to the number of available processors.
`jdk.virtualThreadScheduler.maxPoolSize`	The maximum number of platform threads available to the scheduler. It defaults to 256.

Java Thread 생성

Runnable 이용

```
Thread thread = new Thread(Runnable target);
thread.start();
```



3가지 방법

1. Runnable 구현 클래스 선언 및 객체 생성
2. 익명 객체 사용
3. 람다식 이용

Thread 상속

```
public class WorkerThread extends Thread {
  @Override
  public void run() {
      // ...
  }
}

Thread thread1 = new WorkerThread();
thread1.start();


// 익명 자식 객체
Thread thread2 = new Thread() {
    public void run() {
        // ...
    }
}
```

Java Thread API

```
// default : Thread-n (n은 스레드 번호, 0부터 시작)
thread.setName("스레드 이름")

String name = thread.getName();

Thread thread = Thread.currentThread();

// 1(가장 낮음) ~ 10(가장 높음). default : 5
// Thread 클래스 상수 : MAX_PRIORITY 등
thread.setPrioirty(int 우선순위);
int priority = thread.getPriority();

// 데몬 스레드. 주 스레드가 종료되면 자동 종료
// 반드시 thread.start() 전에 데몬 설정
thread.setDaemon(true);
boolean daemon = thread.isDaemon();
```

CF. 일반 스레드는 main 스레드가 종료되더라도 계속 실행 상태로 남아있다.

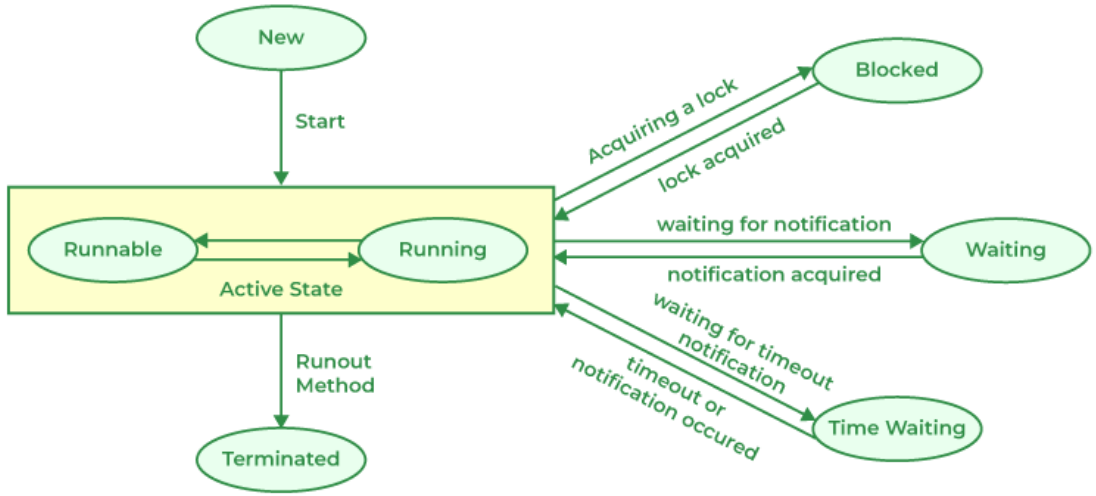
Java Thread Status

상태	열거 상수	설명
객체 생성	NEW	스레드 객체가 생성, 아직 start() 메소드가 호출되지 않은 상태
실행 대기	RUNNABLE	실행 상태로 언제든지 갈 수 있는 상태
일시 정지	WAITING	다른 스레드가 통지할 때까지 기다리는 상태
	TIMED_WAITING	주어진 시간 동안 기다리는 상태
	BLOCKED	사용하고자 하는 객체의 락이 풀릴 때까지 기다리는 상태
종료	TERMINATED	실행을 마친 상태

```
Object.wait with no timeout
Thread.join with no timeout
```

```
Thread.sleep
Object.wait with timeout
Thread.join with timeout
```

```
Thread.State state = thread.getState();

public enum Stats {
    NEW, RUNNABLE, BLOCKED, WAITING,
    TIMED_WAITING, TERMINATED;
}
```

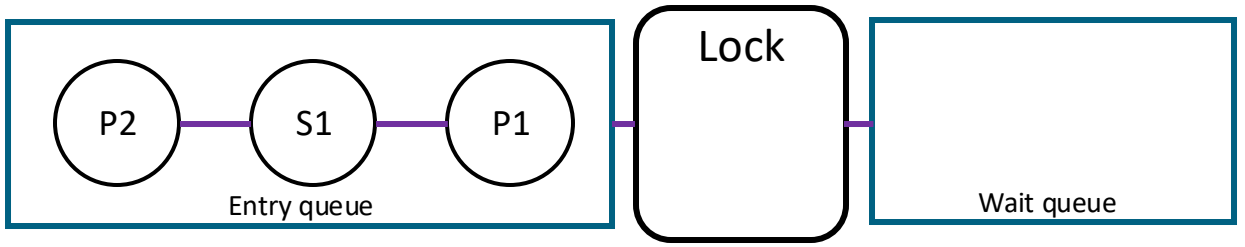


Java Thread 상태 제어

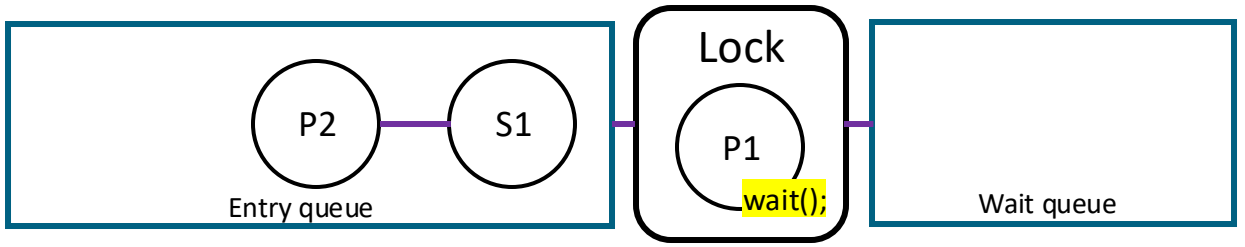
메소드	설명
interrupt()	일시정지 상태의 스레드에서 InterruptedException 발생
wait(), wait(long ms), wait(long ms, int nano)	동기화 블록 내에서 스레드를 일시정지 상태로 만든다.
notify(), notifyAll()	wait() 메소드에 의해 일시정지 상태에 있는 스레드를 실행 대기 상태로 만든다.
join(), join(long ms), join(long ms, int nano)	타겟 스레드가 종료되거나 시간이 지날 때까지 일시 정지
sleep(long ms), sleep(long ms, int nano)	
yield()	실행 중에 우선순위가 동일한 다른 스레드에게 실행을 양보하고 실행 대기 상태가 된다.

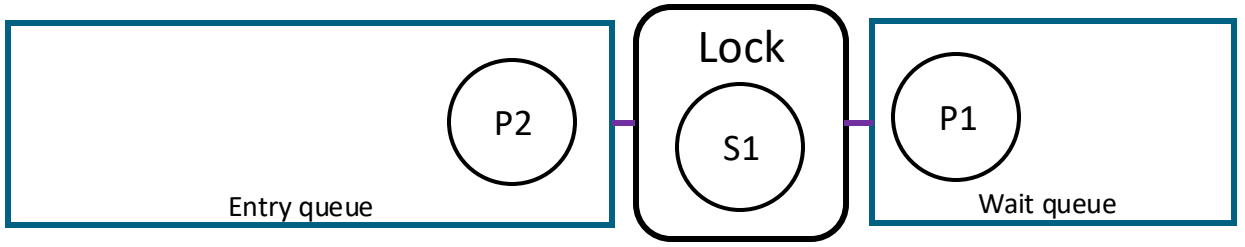
Java Monitors

```
public synchronized void insert(E item) {
  while (BUFFER.isFull()) wait();

  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  count++;

  notify();
}

public synchronized E remove() {
  while(BUFFER.isEmpty()) wait();

  E item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;

  notify();
  return item;
}
```



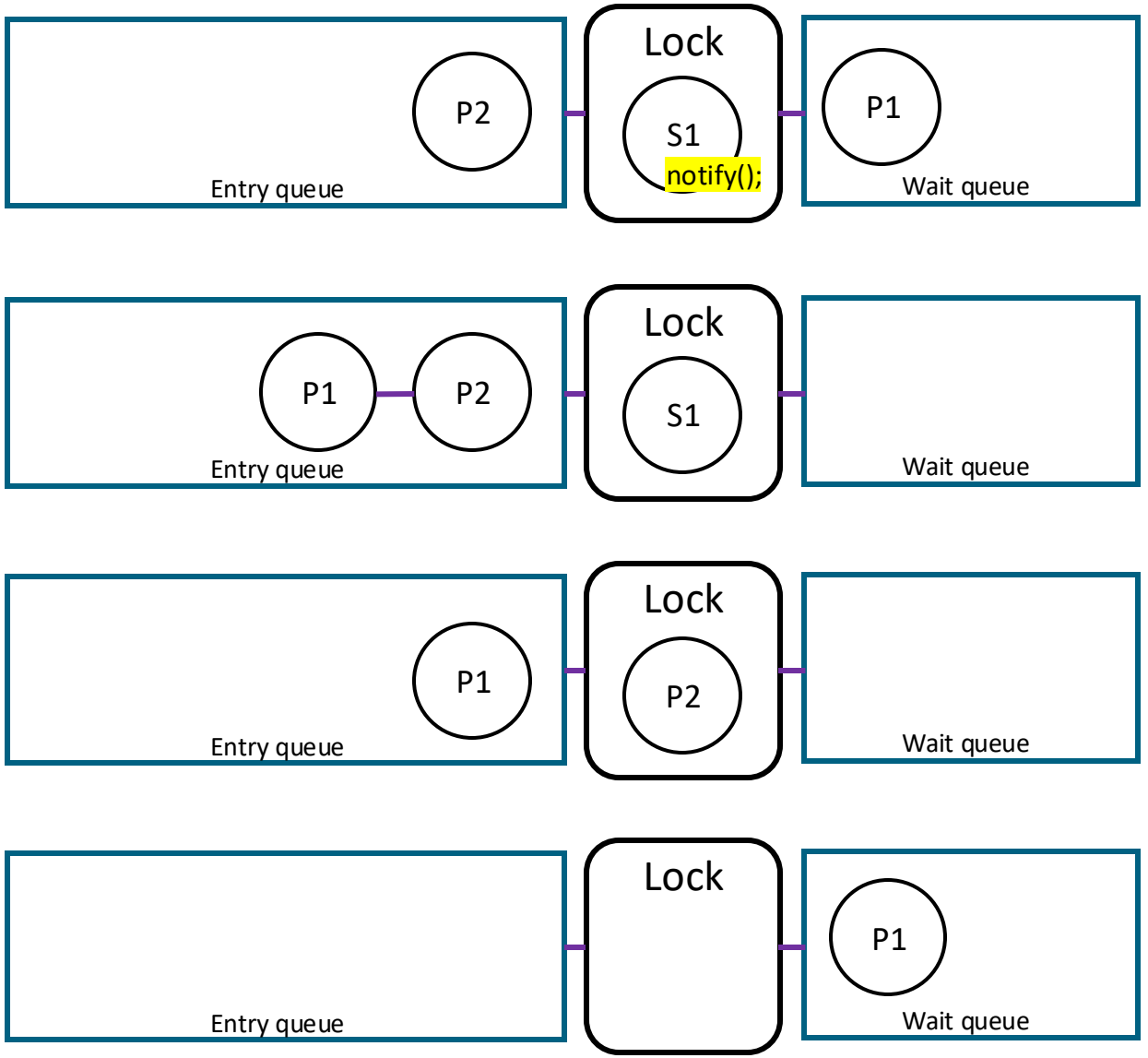
가정: BUFFER is Full





Java Monitors

```
public synchronized void insert(E item) {
  while (BUFFER.isFull()) wait();

  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  count++;

  notify();
}

public synchronized E remove() {
  while(BUFFER.isEmpty()) wait();

  E item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;

  notify();
  return item;
}
```



Yielding, Sleeping 주의할 점

Thread.yield() : CPU 포기하기

```
public synchronized void method1() {
  while (true) {
    Thread.yield();
  }
}
```

```
public synchronized void method1() {
  try {
      Thread.sleep(3000);
  } catch (InterruptedException e) {}
}
```

The thread does not lose ownership of any monitors.

| CPU는 양보하지만 Lock은 양보 못해

Note 2

Java Thread Pool

ExecutorService

메소드명	초기 스레드 수	코어 스레드 수	최대 스레드 수
newCachedThreadPool()	0	0	Integer.MAX_VALUE
newFixedThreadPool(int nThreads)	0	nThreads	nThreads

초기 스레드 수 : ExecutorService 객체가 생성될 때 기본적으로 생성되는 스레드 수

코어 스레드 수 : 사용되지 않는 스레드를 제거할 때 최소한 유지해야 할 스레드 수

최대 스레드 수 : 스레드풀에서 관리하는 최대 스레드 수

```
ExecutorService executorService1 = Executors.newCachedThreadPool();

ExecutorService executorService2 = Executors.newFixedThreadPool(
        Runtime.getRuntime().availableProcessors()
);
```

CPU 코어 수
ex) Apple M1 = 8코어

ExecutorService

```
ExecutorService threadPool = new ThreadPoolExecutor(
        3,    // 코어 스레드 개수
        100,  // 최대 스레드 개수
        120L, // 놀고 있는 시간
        TimeUnit.SECONDS, // 놀고 있는 시간 단위
        new SynchronousQueue<>() // 작업 큐
);
```

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor( corePoolSize: 0, Integer.MAX_VALUE,
            keepAliveTime: 60L, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>());
}
```

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
            keepAliveTime: 0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>());
}
```

ExecutorService

```
ExecutorService threadPool = new ThreadPoolExecutor(
        1,
        2,
        0L,
        TimeUnit.SECONDS,
        new SynchronousQueue<>()
);

Runnable runnable = () -> {
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    System.out.println(Thread.currentThread().getName());
};

threadPool.submit(runnable);
threadPool.submit(runnable);
threadPool.submit(runnable);
```

SynchronousQueue
저장공간 없음.
소비자가 가져갈 때까지 블록

```
Exception in thread "main" java.util.concurrent.RejectedExecutionException
    at threadpool.ExecutorServiceTest.main(ExecutorServiceTest.java:30)
```

ExecutorService

주의
애플리케이션을 종료하려면 스레드풀을 종료시켜야 한다.

리턴 타입	메소드명(매개 변수)	설명
void	shutdown()	남은 모든 작업 처리 후 종료
List<Runnable>	shutdownNow()	처리 중인 스레드에 interrupt 시도 리턴값은 미처리된 작업 목록
boolean	awaitTermination(long timeout, TimeUnit unit)	shutdown() 호출 이후, timeout 내에 완료했는가? 남은 작업은 interrupt

Task 생성

Runnable

```
Runnable runnable = new Runnable() {
        @Override
        public void run() {
            // do something
        }
    };
```

Callable

```
Callable<String> callable = new Callable<>() {
    @Override
    public String call() throws Exception {
        return "";
    }
};
```

Task 요청

리턴 타입	메소드명(매개 변수)	설명
void	execute(Runnable command)	처리 중 예외 발생 시 스레드 종료 및 제거
Future<?> Future<V> Future<V>	submit(Runnable task) submit(Runnable task, V result) submit(Callable<V> task)	예외 발생 시에도 종료하지 않고 재사용 가급적 submit 사용

```
ExecutorService executorService = Executors.newFixedThreadPool(2);
Future<String> future = executorService.submit(new Callable<>() {
            @Override
            public String call() {
                return "hi";
            }
        });
```

blocking 작업 통보 + Future API

리턴 타입	메소드명(매개 변수)	설명
V	get()	작업 완료 시까지 블로킹 및 결과(V) 리턴
V	get(long timeout, TimeUnit unit)	작업 완료 시까지 블로킹 및 결과(V) 리턴 timeout 시 TimeoutException 발생
boolean	cancel(boolean mayInterruptIfRunning)	작업 처리가 진행 중일 경우 취소시킴 (이미 완료 등 취소할 수 없으면 false)
boolean	isCancelled()	작업이 취소되었는지 여부 (완료되기 전에 취소된 경우만 true)
boolean	isDone()	작업 처리가 완료되었는지 여부 (완료 = 정상적, 예외, 취소 등)

blocking 작업 통보 + Future API

submit(Runnable task)

```
Future<?> future = executorService.submit(
    new Runnable() {
        @Override
        public void run() { ... }
    }
);
Object result = future.get(); // null 리턴
```

submit(Callable<V> task)

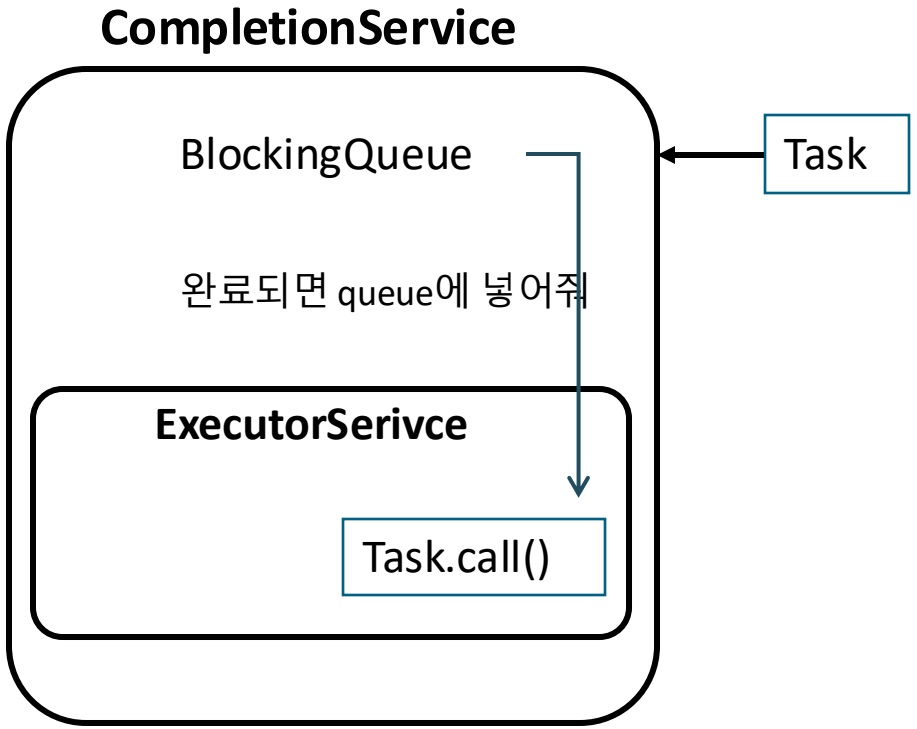
```
Future<String> future = executorService.submit(
    new Callable<>() {
        @Override
        public String call() { ... }
    }
);
String result = future.get();
```

submit(Runnable task, V result)

```
Result result = new Result();
Runnable task = new Task(result);
Future<Result> future = executorService.submit(task, result);
result = future.get();

class Task implements Runnable {

    Result result;

    public Task(Result result) {
        this.result = result;
    }

    @Override
    public void run() {
        // ...
        result.addResult(..);
    }
}
```

blocking 작업 통보 + Future API

작업 완료 순으로 통보받기




```
protected void done() { completionQueue.add(task); }
```

```
ExecutorCompletionService<String> completionService
        = new ExecutorCompletionService<>(executorService);

completionService.submit(new Callable<>() {
    @Override
    public String call() {
        return "hi";
    }
});

Future<String> future = completionService.poll(); // can null
Future<String> future2 = completionService.take(); // blocking
```

```
public Future<V> take() throws InterruptedException {
    return completionQueue.take();
}

public Future<V> poll() {
    return completionQueue.poll();
}
```

callback 방식 1 : callback class

```java
public class Task implements Runnable {

    private int time;
    private Callback callback;

    public Task(int time, Callback callback) {
        this.time = time;
        this.callback = callback;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {}
        callback.callback(Thread.currentThread().getName());
    }
}
```

```java
public class Callback {

    public synchronized void callback(String threadName) {
        System.out.println(threadName);
    }
}
```

```java
public static void main(String[] args) throws InterruptedException {

    ExecutorService executorService = Executors.newFixedThreadPool(3);

    Callback callback = new Callback();
    Task task1 = new Task(1500, callback);
    Task task2 = new Task(2000, callback);
    Task task3 = new Task(1000, callback);

    executorService.submit(task1);
    executorService.submit(task2);
    executorService.submit(task3);

    executorService.shutdown();
}
```

callback 방식 2 : CompletableFuture

```
// 스레드풀 생성
ExecutorService executorService = Executors.newFixedThreadPool(3);

// 비동기 작업을 스레드풀에 제출하고 콜백 처리
CompletableFuture.supplyAsync(
                () -> {
                    // 비동기 작업 수행
                    return null;
                }, executorService)
        .thenAccept(result -> {
            // 작업 완료 후 콜백 처리
        });
```

```
// 스레드풀 종료
executorService.shutdown();
try {
    // 스레드풀의 모든 작업이 완료될 때까지 대기
    executorService.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

이외에도
java.nio.channels.CompletionHandler
존재