

- 스레드 종료 규칙 - 한 스레드에서 `Thread.join()`을 호출하면 `join` 대상 스레드의 모든 작업은 `join()` 반복 후 작업한다 happens-before 관계를 가진다.
  - ↳ `Thread.join()` 호출전, `Thread`의 모든 작업이 완료되어야 하며 이 작업은 `join()` 한환 이후에 참조 가능
- 1 ~ 100 까지 더하는 sumTask 예제.
- 인터럽트 규칙 - 한 스레드에서 `Thread.interrupt()`을 호출하는 작업이 인터럽트된 스레드가 인터럽트를 강제하는 시점에 작업한다 happens-before 관계가 성립한다.
- 객체 생성 규칙 - 객체의 생성자는 마지막에 생성된 이후에만 다른 스레드에 의해 참조되도록 보장한다.
- 면티액 규칙 - 한 스레드에서 `synchronized` 블록 종료후, 그 면티액을 얻는 모든 스레드는 해당 블록 내의 모든 작업을 볼 수 있다.
- 전이 규칙 - 만약 A가 B보다 happens-before 관계에 있고 B가 C보다 happens-before 관계에 있다면, A는 C보다 happens-before 관계에 있다.

## 〈비밀리 가시성 part 정리〉

Volatile, 블록 동기화 기법 (`synchronized`, `ReentrantLock`) 사용시 비밀리 가시성 문제가 발생하지 않는다.

6장

## 동기화 - synchronized

↳ 공유자원 (ex) 행변수

↳ 동시성 문제 : 멀티스레드 환경에서 같은 자원에 여러 스레드가 동시에 접근할 때 발생하는 문제.

공유자원에 대한 접근을 적절하게 동기화해서 동시성 문제가 발생하지 않게 방지하는 것이 중요!

```
package thread.sync;

public interface BankAccount {
    boolean withdraw(int amount);
    int getBalance();
}

@Override
public boolean withdraw(int amount) {
    ① 체크 → ② 충금
    log("거래 시작: " + getClass().getSimpleName());
    log("[검증 시작] 출금액: " + amount + ", 잔액: " + balance);
    if (balance < amount) {
        log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);
        return false;
    }
    log("[검증 완료] 출금액: " + amount + ", 변경 잔액: " + balance);
    sleep(1000); // 출금에 걸리는 시간으로 가정
    balance = balance - amount;
    log("거래 종료");
    return true;
}
```

주의사항: withdraw 메서드

true: 계좌잔액 > 출금액  
false: 계좌잔액 < 출금액

```
public static void main(String[] args) throws InterruptedException {
    BankAccount account = new BankAccountV1(1000); // 초기잔액 1000원

    Thread t1 = new Thread(new WithdrawTask(account, 800), "t1");
    Thread t2 = new Thread(new WithdrawTask(account, 800), "t2");
    t1.start();
    t2.start();

    sleep(500); // 검증 완료까지 잠시 대기
    log("t1 state: " + t1.getState());
}
```

t1, t2 각 스레드에서 800원 출금 시도

```
package thread.sync;
import java.util.concurrent.TimeUnit;

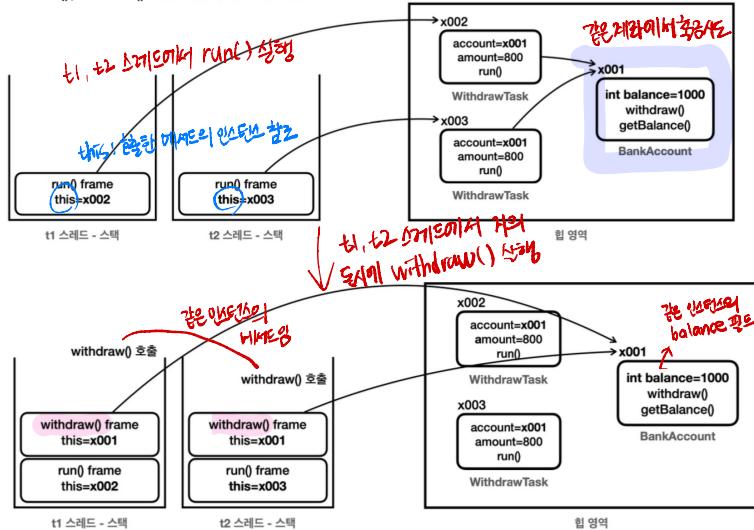
public class WithdrawTask implements Runnable {
    private BankAccount account;
    private int amount;

    public WithdrawTask(BankAccount account, int amount) {
        this.account = account;
        this.amount = amount;
    }

    @Override
    public void run() {
        account.withdraw(amount);
    }
}
```

↳ 공유상당 Runnable 구현체

t1.start(), t2.start() 호출 직후의 메모리 그림



```
log("t2 state: " + t2.getState());
t1.join(); // main 스레드가
t2.join(); // 최종 잔액 확인
log("최종 잔액: " + account.getBalance()); // 잔액은 200원이 될
}
```

Output: -600

↑ (t1, t2 인스턴스 동적 실행)

## 동시성 문제

예시) 악성유저가 2대의 PC에서 동시에 같은 계좌의 돈을 충금한다고 가정

t1 스레드가 약간의 차이로 먼저 실행되고 나서 t2 스레드 실행됨다고 가정

초기 계좌 잔액: 1000 원

↓ t1 충금 800원

잔액: 200원

↓ t2 충금 800원

Expected: 잔액보다 충금금액이 크므로 충금에 실패해야 하는데...?

Real: 실행 결과에 따르면接地气하게 t1, t2는 각각 900원씩 총 1600원 충금에 성공한다.

↳ 마지막 잔액은 -600원

왜 이런 문제가 발생했을까?

계좌 출금시 잔고 체크 로직

```
if (balance < amount) {
    log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);
    return false;
}
```

balance에 volatile 키워드를 사용하여 해결 불가  
volatile은 한 스레드가 값을 변경했을 때 다른 스레드에서  
변경된 값을 즉시 볼 수 있게 하는 버려지기 시스템을 해결 가능.

① t1, t2 순서로 실행 가정

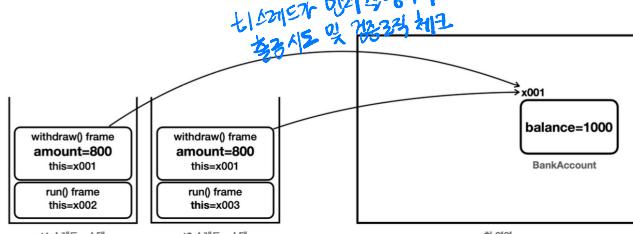
```
withdraw { t2
    if (balance < amount) { t1
        return false;
    }
    sleep(1000);
    balance = balance - amount;
}
```

t1: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.

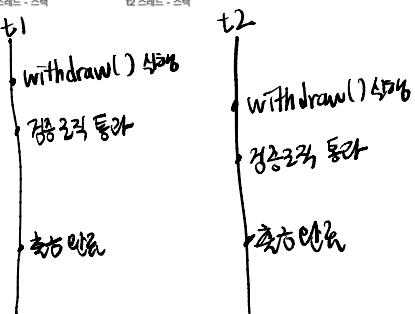
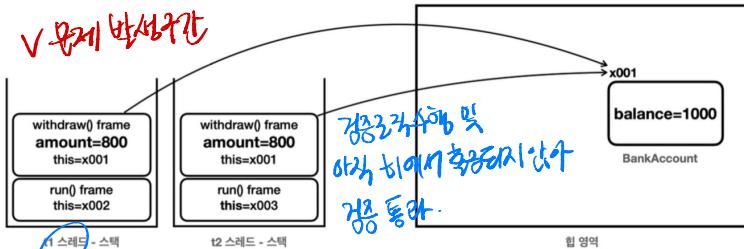
②

```
withdraw {
    if (balance < amount) { t2
        return false;
    }
    sleep(1000); t1
    balance = balance - amount;
}
```

t2: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.



✓ 문제 발생구간



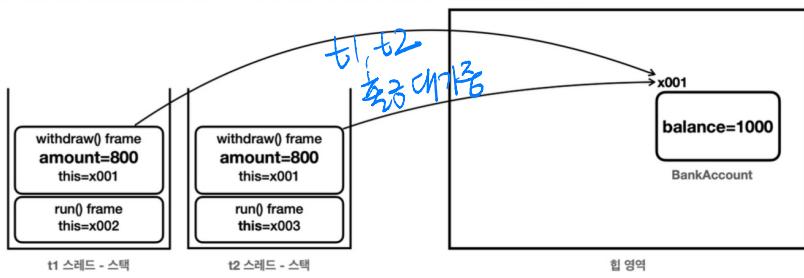
③

```

withdraw {
    if (balance < amount) {
        return false;
    }
    sleep(1000); t1 t2
    balance = balance - amount;
}

```

t1: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.  
t2: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.



④

```

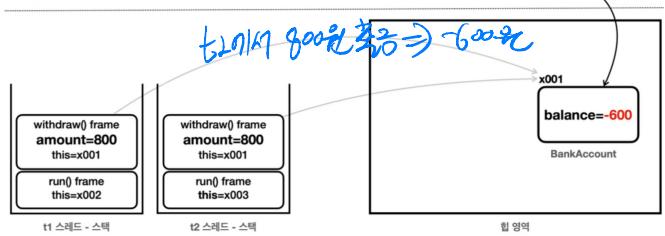
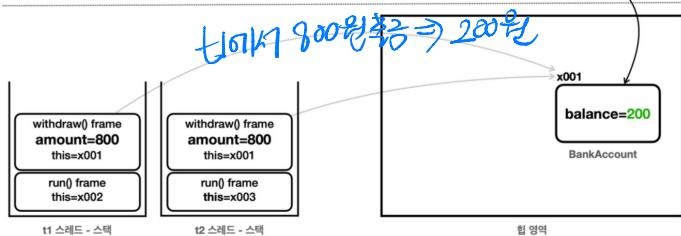
withdraw {
    if (balance < amount) {
        return false;
    }
    sleep(1000);
    balance = balance - amount; t1
}

```

```

withdraw {
    if (balance < amount) {
        return false;
    }
    sleep(1000);
    balance = balance - amount; t2
}

```



Result : t1, t2 둘째 모두 800원 축감 성공

$$\text{최종잔액: } 1000 - 800 \times 2 = -600\text{원}$$

$t_1, t_2$  가 안전히 동시에 실행될 때

```

withdraw {
    if balance < amount:  $\leftarrow t_1 \wedge t_2$  t1: 잔액(1000) > 축출액(800)
        return false
    sleep(1000)  $\leftarrow t_1 \wedge t_2$  흡금 대기중
    balance -= amount
}
 $\leftarrow t_1, t_2$  t1: balance(200) = balance(1000) - amount(800)
 $\leftarrow t_1, t_2$  t2: balance(200) = balance(1000) - amount(800)

```

따라서, 최종 잔액 (balance)은 200원이 된다.

$$\text{balance} = \underline{\text{balance} - \text{amount}}$$

① 두 변수의 값 조회 -  $t_1, t_2$  두 스레드가 동시에 X001. balance의  
 값을 읽음(1000)  
 ② 계산  
 두 스레드는 1000 - 800 을 계산

③ 저장  
 두 스레드는 balance에 200 저장

### 임계영역

① 깜빡 → ② 흡금

근본 원인: 공유 자원을 여러 단계에 걸쳐 사용하기 때문

```

출금() {
    1. 검증 단계: 잔액(balance) 확인
    2. 출금 단계: 잔액(balance) 감소
}

```

→ 근처에 있는 하나의 큰 가방  
 내가 사용하는 값이 중간에 변경되지 않을 것이라는 가정  
 → 공유자원으로 중간에 다른 스레드에 의해  
 값이 변경될 수 있다.

→ withdraw() 메서드를 한 번에 하나의 스레드만 실행

$t_1$ 이 먼저 흡금() 호출하고 처음부터 끝까지 양수하고 나서 그 다음  $t_2$  스레드가 처음부터 끝까지 축출() 메서드 양수

공유자원인 balance를 한 번에 하나의 스레드만 변경 가능 → 공간에 값이 바뀌지 않음

## 임계 영역 (critical section)

- 여러 스레드가 동시에 접근하면 데이터 복일치, 예상치 못한 동작이 발생할 수 있는 코드 부분
- ↳ 공유 변수나 공유 객체를 수정할 때, 앞 예제의 withdraw()

어떻게 한 번에 하나의 스레드만 접근할 수 있도록 임계 영역을 보호할 수 있을까?  
↳ Java에서는 synchronized 키워드 사용

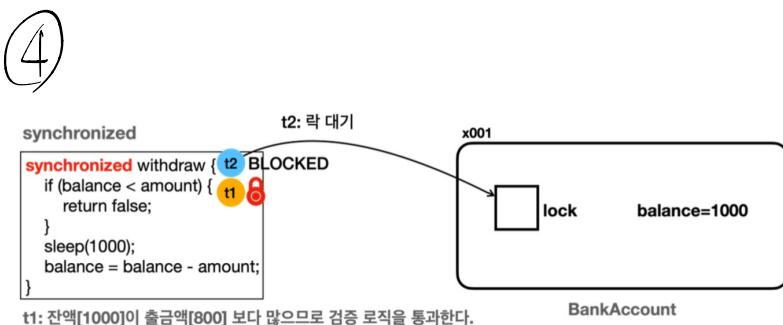
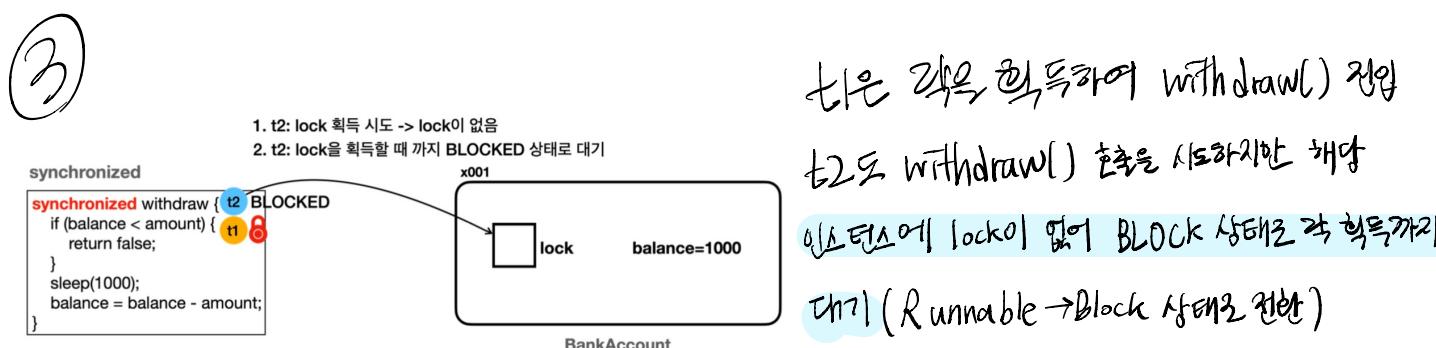
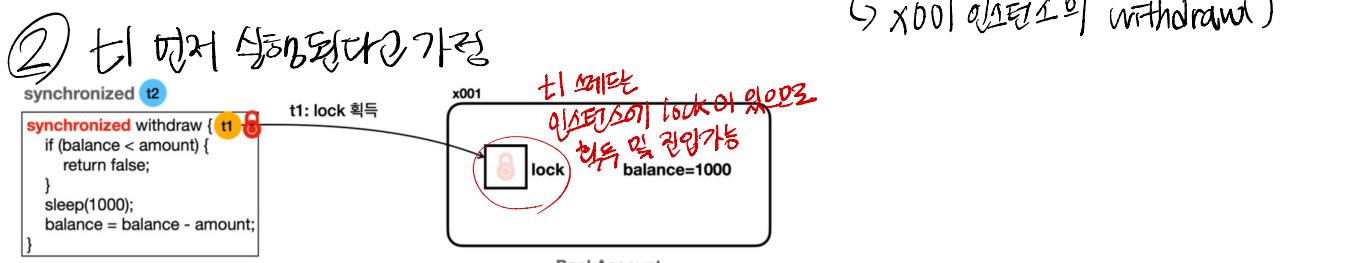
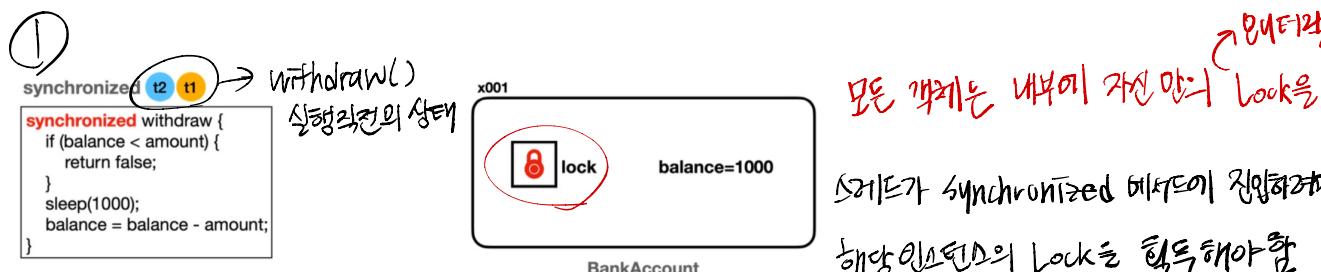
## Synchronized : 한 번에 하나의 스레드만 실행 가능

```
@Override  
public synchronized boolean withdraw(int amount) {  
    log("거래 시작: " + getClass().getSimpleName());  
  
    log("[검증 시작] 출금액: " + amount + ", 잔액: " + balance);  
    if (balance < amount) {  
        log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);  
        return false;  
    }  
    log("[검증 완료] 출금액: " + amount + ", 잔액: " + balance);  
    sleep(1000);  
    balance = balance - amount;  
    log("[출금 완료] 출금액: " + amount + ", 변경 잔액: " + balance);  
  
    log("거래 종료");  
    return true;  
}
```

Synchronized 추가 후 결과

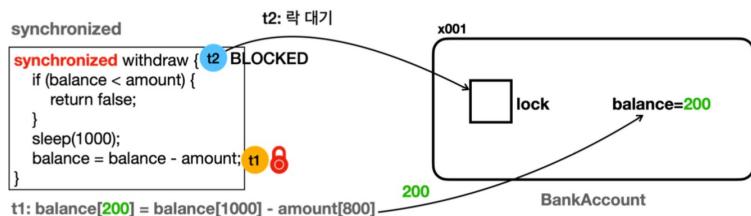
↳ balance: 200 원

```
@Override  
public synchronized int getBalance() {  
    return balance;  
}
```

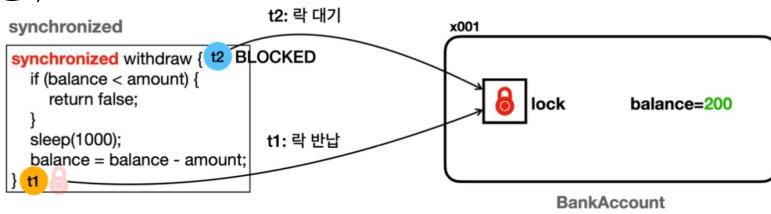


t1: 800원 출금 및 잔액 200원

t2: 대기



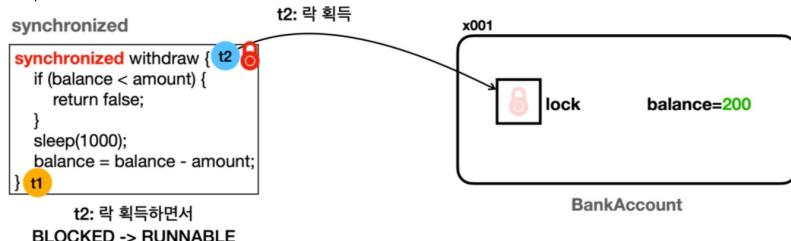
(6)



t1: 락 반납

t2: 락 대기

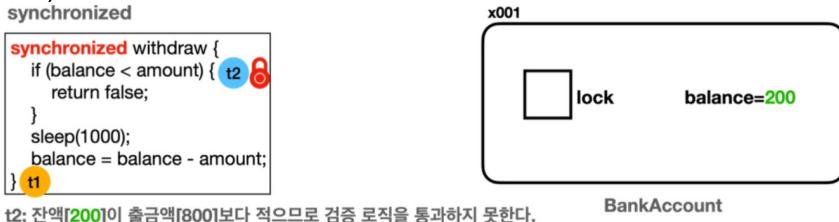
(7)



t1: 종료

t2: 락 획득 (runnable 됨)

(8)



balance(200) &lt; 출금액[800]

이므로 검증로직 통과 실패

(9) t2는 락을 반납하면서 return

V 인스턴스의 락을 땔는 순서는

result: 최종잔액: 200원 (t1 성공 t2 실패) 보장되지 않는다.

Synchronized는 한번에 하나의 스레드만 실행가능하다는 정때문에 여러 스레드가 동시에 실행할 수 없음  $\Rightarrow$  전체적으로 성능 저하  
따라서, 필요한 구간만 딱 한정해서 설정하는 것이 좋다.

```

public synchronized boolean withdraw(int amount) {
    log("거래 시작: " + getClass().getSimpleName());

    log("[검증 시작] 출금액: " + amount + ", 잔액: " + balance);
    if (balance < amount) {
        log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);
        return false;
    }
    log("[검증 완료] 출금액: " + amount + ", 잔액: " + balance);
    sleep(1000);
    balance = balance - amount;
    log("[출금 완료] 출금액: " + amount + ", 변경 잔액: " + balance);

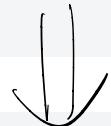
    log("거래 종료");
    return true;
}

```

양제 구역

↓  
synchronized 를 특정 코드 블록에

획득화해서 적용 가능



```

@Override
public boolean withdraw(int amount) {
    log("거래 시작: " + getClass().getSimpleName());
    작을 획득할 인스턴스의 참조
    synchronized (this) {
        log("[검증 시작] 출금액: " + amount + ", 잔액: " + balance);
        if (balance < amount) {
            log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);
            return false;
        }
        log("[검증 완료] 출금액: " + amount + ", 잔액: " + balance);
        sleep(1000);
        balance = balance - amount;
        log("[출금 완료] 출금액: " + amount + ", 변경 잔액: " + balance);
    }

    log("거래 종료");
    return true;
}

```

양제구역에만  
기여드 적용

자바의 동기화는 여러 스레드가 동시에 접근할 수 있는 자원(객체, 메서드)에 대해 일관성 있고  
안전한 접근을 보장하기 위한 막기나옴

*synchronized* — 메서드 동기화  
↓  
블록 동기화

⇒ race condition, 데이터 일관성 문제  
해결 가능