동기화 - syncrhonized

소유자	종수 김
태그	

출금 예제 - 시작

멀티 쓰레드를 사용할 때 가장 주의해야 할 점은, 같은 자원(리소스)에 **여러 쓰레드가 동시에 접근할 때 발생**하는 '동시성 문제'

공유자원 : 여러 쓰레드가 접근하는 자원
Ex) 인스턴스의 필드(멤버 변수)

이런 공유 자원에 대한 접근을 적절하게 '**동기화(synchronization)**'하여 동시성 문제가 발생하지 않게 방지해야 함.

은행 출금 예제

```
package thread.sync;

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

- `BankAccount` 인터페이스이다. 앞으로 이 인터페이스의 구현체를 점진적으로 발전시키면서 문제를 해결할 예
  정이다.
- `withdraw(amount)` : 계좌의 돈을 출금한다. 출금할 금액을 매개변수로 받는다.
  - 계좌의 잔액이 출금할 금액보다 많다면 출금에 성공하고, `true` 를 반환한다.
  - 계좌의 잔액이 출금할 금액보다 적다면 출금에 실패하고, `false` 를 반환한다.
- `getBalance()` : 계좌의 잔액을 반환한다.

```
package thread.sync;

import static util.MyLoggerThread.log;
```

```
import static util.ThreadUtils.sleep;

public class BankAccountV1 implements BankAccount{
    private int balance;

    public BankAccountV1(int initialBalance) {
        this.balance = initialBalance;
    }

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

        // 잔고가 출금액 보다 적으면 진행하면 안됨.
        log("[검증 시작] 출금액 : " + amount + ", 잔액: " + balance);
        if(balance < amount) {
            log("[검증 실패]");
            return false;
        }

        // 잔고가 출금액 보다 많으면 진행
        log("[검증 완료] 출금액 : " + amount + ", 잔액: " + balance);
        sleep(1000); // 출금에 걸리는 시간으로 가정
        balance -= amount;

        log("[출금 완료] 출금액 : " + amount + ", 잔액: " + balance);

        log("거래 종료");
        return true;
    }

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

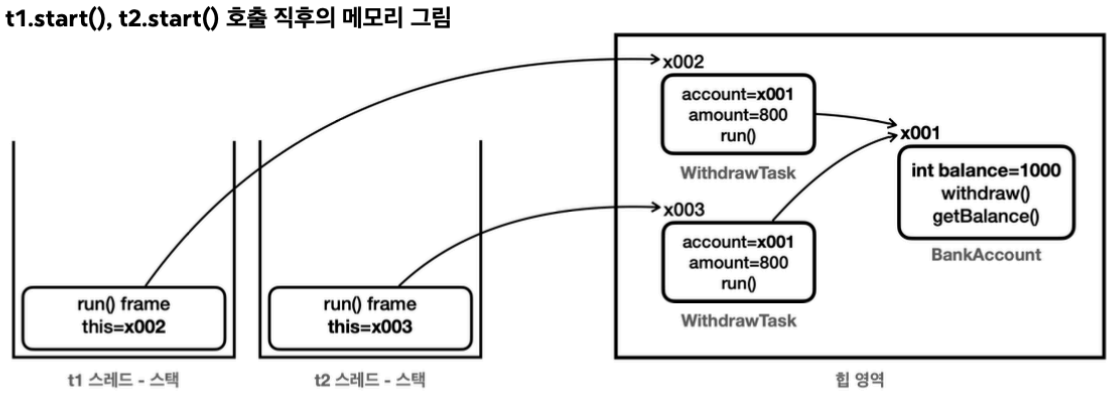
- `BankAccountV1` 은 `BankAccount` 인터페이스를 구현한다.

- 생성자를 통해 계좌의 초기 잔액를 저장한다.
- `int balance` : 계좌의 잔액 필드
- `withdraw(amount)` : 검증과 출금 2가지 단계로 나누어진다.
    - **검증 단계**: 출금액과 잔액을 비교한다. 만약 출금액이 잔액보다 많다면 문제가 있으므로 검증에 실패하고,`false` 를 반환한다.
    - **출금 단계**: 검증에 통과하면 잔액이 출금액보다 많으므로 출금할 수 있다. 잔액에서 출금액을 빼고 출금을완료하면, 성공이라는 의미의 `true` 를 반환한다.
- `getBalance()` : 잔액을 반환한다.

동시성 문제 코드

```
package thread.sync;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class BankMain {
    public static void main(String[] args) throws InterruptedException {
        BankAccount account = new BankAccountV1(1000);

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

        sleep(500); // 검증 완료까지 잠시 대기
        log("t1 state: " + t1.getState());
        log("t2 state: " + t2.getState());

        t1.join(); // 최종 잔액 확인을 위한, 쓰레드 종료 대기
        t2.join();

        log("최종 잔액 : " + account.getBalance());
    }
}
```

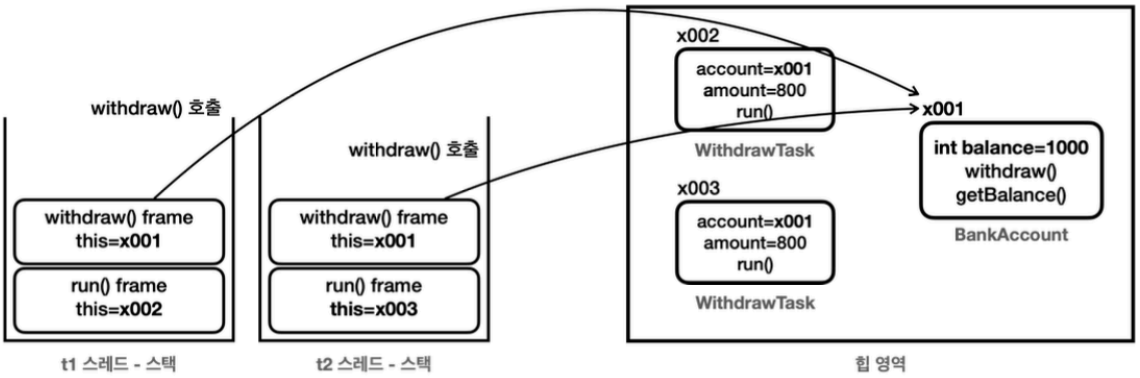
```
23:06:04.418 [       t2] 거래 시작 : BankAccountV1
23:06:04.418 [       t1] 거래 시작 : BankAccountV1
23:06:04.423 [       t1] [검증 시작] 출금액 : 800, 잔액: 1000
23:06:04.423 [       t2] [검증 시작] 출금액 : 800, 잔액: 1000
23:06:04.423 [       t1] [검증 완료] 출금액 : 800, 잔액: 1000
23:06:04.423 [       t2] [검증 완료] 출금액 : 800, 잔액: 1000
23:06:04.899 [     main] t1 state: TIMED_WAITING
23:06:04.899 [     main] t2 state: TIMED_WAITING
23:06:05.426 [       t2] [출금 완료] 출금액 : 800, 잔액: 200
23:06:05.426 [       t2] 거래 종료
23:06:05.428 [       t1] [출금 완료] 출금액 : 800, 잔액: -600
23:06:05.429 [       t1] 거래 종료
23:06:05.431 [     main] 최종 잔액 : -600
```



- 각각의 쓰레드의 스택에서 run()이 실행 됨.
- t1 쓰레드는 WithdrawTask(x002) 인스턴스의 run()을 실행함.
- t2 쓰레드는 WithdrawTask(x003) 인스턴스의 run()을 실행함.

> t1, t2 쓰레드 모두, 동일한 참조(BankAccount[x001])를 바라보고 있음.

- 스택 프레임의 this에는 호출한 메서드의 인스턴스 참조가 들어있다.
- 두 쓰레드는 같은 계좌(x001)에 대해서 출금을 시도한다.



- t1 쓰레드의 run()에서 withdraw()를 실행한다.
- 거의 동시에 t2 쓰레드의 run()에서 withdraw()를 실행한다.
- t1 쓰레드와 t2 쓰레드는 같은 BankAccount(x001) 인스턴스의 withdraw() 메서드를 호출한다.
- 따라서, 두 쓰레드는 같은 BankAccount(x001) 인스턴스에 접근하고, 또 x001 인스턴스에 있는 잔액(balance) 필드도 함께 사용한다.

실행 결과

```
11:09:40.185 [       t1] 거래 시작: BankAccountV1
11:09:40.185 [       t2] 거래 시작: BankAccountV1
11:09:40.192 [       t1] [검증 시작] 출금액: 800, 잔액: 1000
11:09:40.192 [       t2] [검증 시작] 출금액: 800, 잔액: 1000
11:09:40.192 [       t1] [검증 완료] 출금액: 800, 잔액: 1000
11:09:40.192 [       t2] [검증 완료] 출금액: 800, 잔액: 1000
11:09:40.673 [     main] t1 state: TIMED_WAITING
11:09:40.673 [     main] t2 state: TIMED_WAITING
11:09:41.195 [       t1] [출금 완료] 출금액: 800, 변경 잔액: 200
11:09:41.195 [       t1] 거래 종료
11:09:41.197 [       t2] [출금 완료] 출금액: 800, 변경 잔액: -600
11:09:41.197 [       t2] 거래 종료
11:09:41.200 [     main] 최종 잔액: -600
```

- **참고**: 여기서는 `t1` 스레드가 먼저 실행되었다. 그런데 실행 환경에 따라서 `t1`, `t2`가 완전히 동시에 실행될 수도 있다. 이 경우 출금액은 같고, 잔액은 200원이 된다. 이 부분은 바로 뒤에서 설명한다.

- 완전히 동시에 실행하는 경우도 있을 수 있음.

동시성 문제

위의 시나리오는 악의적인 사용자가 2대의 PC에서 동시에 같은 계좌의 돈을 출금한다고 가정.

- t1, t2 쓰레드는 거의 동시에 실행되었으나, 아주 약간의 차이로 t1 쓰레드가 먼저 실행되고 t2가 실행된다고 가정.
- 처음 계좌의 잔액은 1000원, t1 쓰레드가 800원을 출금하면, 잔액은 200원이 남음.
- 계좌의 잔액은 200원, t2 쓰레드가 800원을 출금하면 잔액보다 더 많은 돈을 출금하게 되므로 출금에 실패해야 함.

> 실행 결과는 기대와 다르게 t1,t2 각각 800원씩 1600원 출금에 성공함.
> 계좌의 잔액은 -600원이 됨

계좌를 출금할 때 잔고를 체크하는 로직이 있는데도 불구하고 왜 이런 문제가 발생할까 ?

> 참고 : volatile은 메모리 가시성 문제를 해결하는 것이므로 지금 상황에서 해결법이 되지 않음.

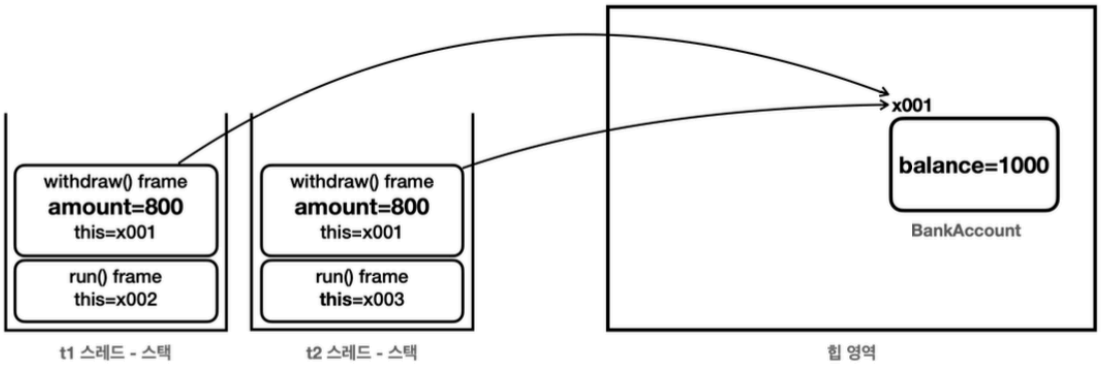
동시성 문제

t1, t2 순서로 실행한다 가정

t1, t2 순서로 실행 가정

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

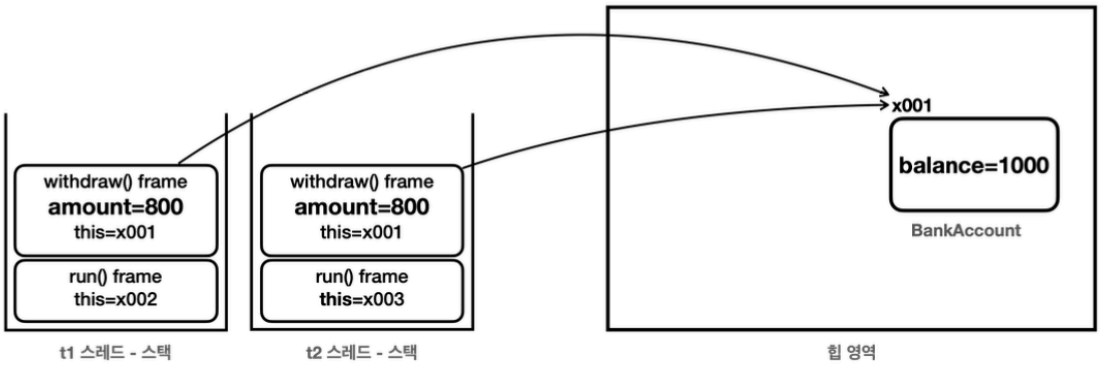
t1: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.



- `t1` 이 약간 먼저 실행되면서, 출금을 시도한다.
- `t1` 이 출금 코드에 있는 검증 로직을 실행한다. 이때 잔액이 출금 액수보다 많은지 확인한다.
  - 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.

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

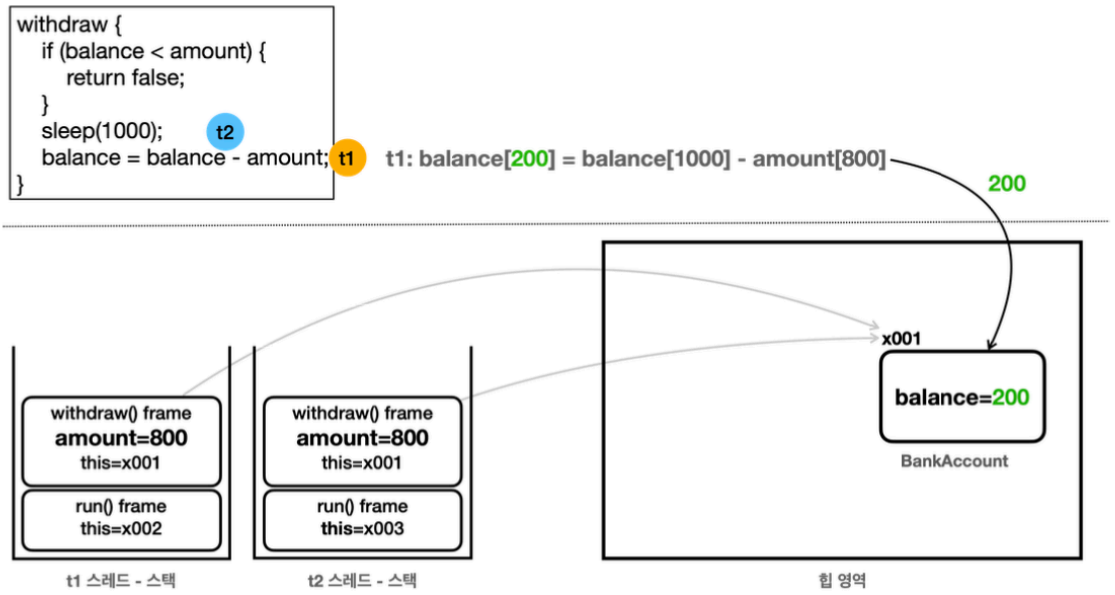
t2: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.



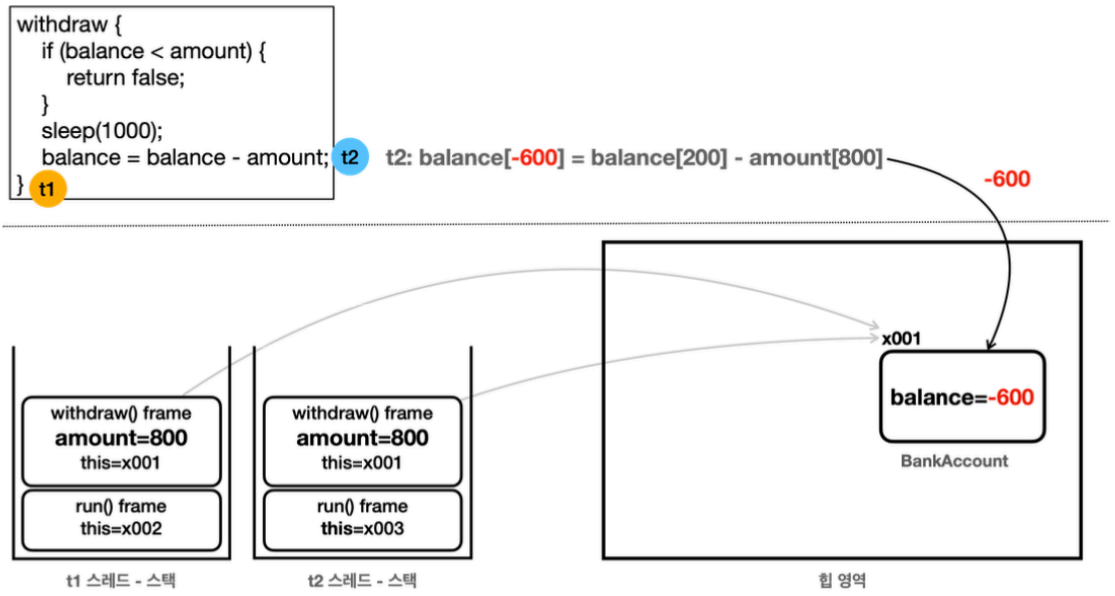
- `t1` : 출금 검증 로직을 통과해서 출금을 위해 잠시 대기중이다. 출금에 걸리는 시간으로 생각하자.
- `t2` : 검증 로직을 실행한다. 잔액이 출금 금액보다 많은지 확인한다.
  - 잔액[1000]이 출금액[800] 보다 많으므로 통과한다.

- **t1이 아직 잔액을 줄이지 못했기 때문에, t2는 검증 로직에서 현재 잔액을 1,000원으로 확인**

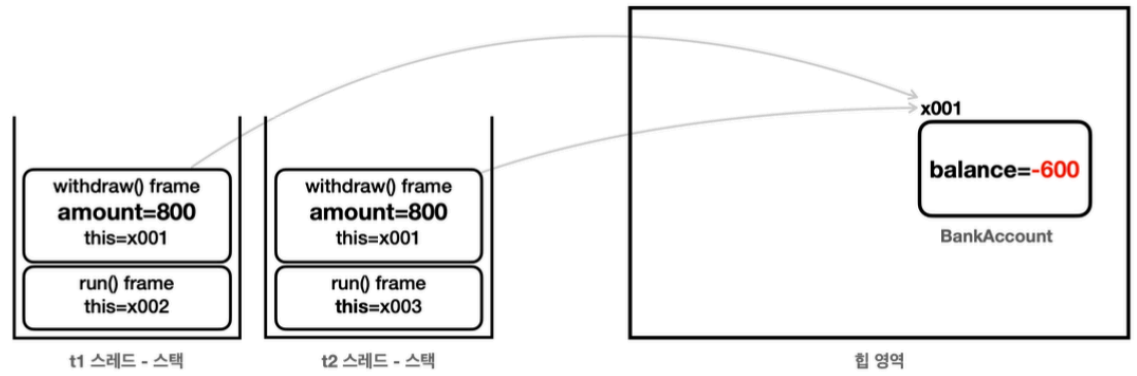
- t1 검증 로직을 통과하고 바로 잔액을 줄였다면 이런 문제가 발생하지 않았을까?
    - sleep(1000)을 제거하더라도, balance -= amount; 가 동시에 실행될 수도 있는 것.
    - sleep(1000)은 단순 확인을 쉽게하는 용도일 뿐.
    - 실제로 '동시에' 접근할 수도 있음.
- t2도 t1과 마찬가지로 로직을 통과하고 잔액을 줄이는 중.



- `t1`은 800원을 출금하면서, 잔액을 1000원에서 출금 액수인 800원 만큼 차감한다. **이제 계좌의 잔액은 200원이 된다.**



- `t2`는 800원을 출금하면서, 잔액을 200원에서 출금 액수인 800원 만큼 차감한다. **이제 잔액은 -600원이 된다.**



결과

- `t1` : 800원 출금 완료
- `t2` : 800원 출금 완료
- 처음 원금은 1000원이었는데, 최종 잔액은 -600원이 된다.
- 은행 입장에서 마이너스 잔액이 있으면 안된다!

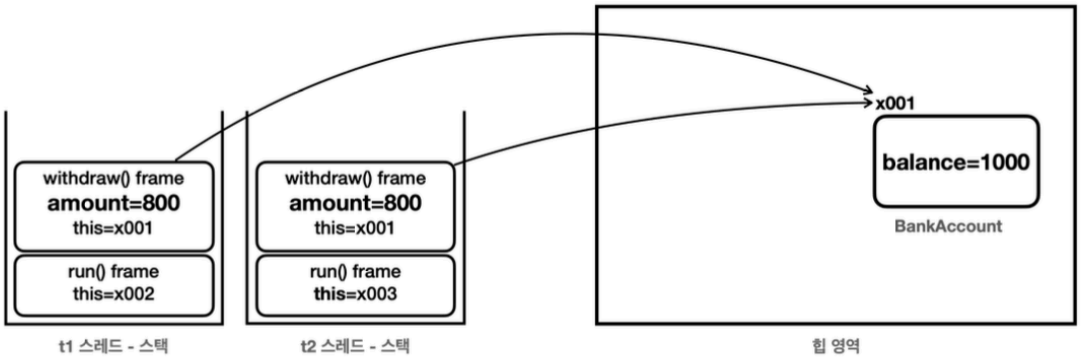
- 같은 인스턴스의 정보를 같이 바라보고 있으며, 같은 '공유 변수'를 수정하기 때문에 생기는 문제

만약 t1,t2가 동시에 실행되었다면 ?

t1, t2 동시에 실행 가정

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

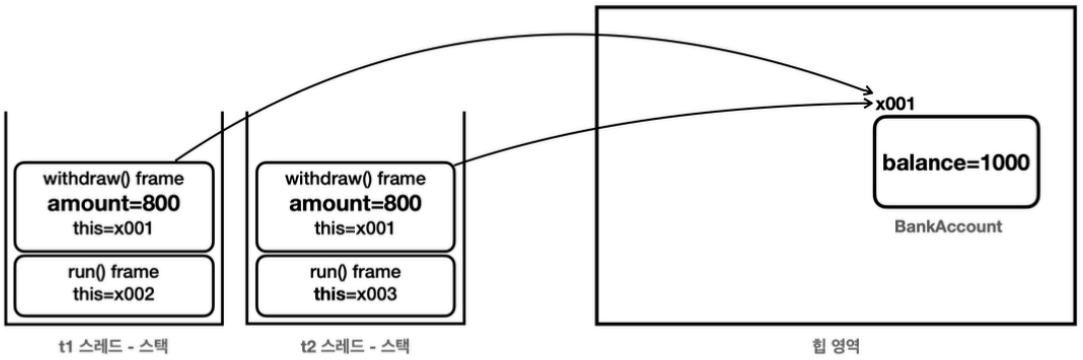
t1: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.
t2: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.

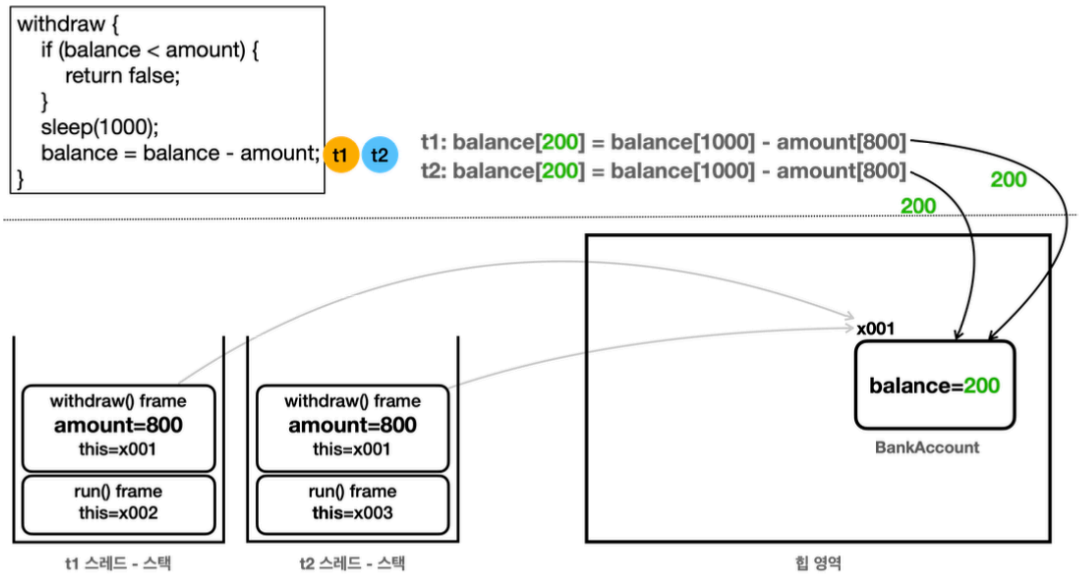


- `t1`, `t2`는 동시에 검증 로직을 실행한다. 잔액이 출금 금액보다 많은지 확인한다.
  - 잔액[1000]이 출금액[800] 보다 많으므로 둘다 통과한다.

```
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.





- `t1`은 800원을 출금하면서, 잔액을 1000원에서 출금 액수인 800원 만큼 차감한다. **이제 잔액은 200원이 된다.**
- `t2`은 800원을 출금하면서, 잔액을 1000원에서 출금 액수인 800원 만큼 차감한다. **이제 잔액은 200원이 된다.**
- `t1`, `t2`가 동시에 실행되기 때문에 둘다 잔액(`balance`)을 확인하는 시점에 잔액은 1000원이다!
- `t1`, `t2`둘다 동시에 계산된 결과를 잔액에 반영하는데, 둘다 계산 결과인 200원을 반영하므로 **최종 잔액은 200원이 된다.**

```
balance = balance - amount;
```

이 코드는 다음의 단계로 이루어진다.

1. 계산을 위해 오른쪽에 있는 `balance` 값과 `amount` 값을 조회한다.
2. 두 값을 계산한다.
3. 계산 결과를 왼쪽의 `balance` 변수에 저장한다.

- 여기서 1번 단계의 `balance` 값을 조회할 때 `t1`, `t2` 두 스레드가 동시에 `x001.balance`의 필드 값을 읽는다. 이때 값은 `1000`이다. 따라서 두 스레드는 모두 잔액을 1000원으로 인식한다.
- 2번 단계에서 두 스레드 모두 `1000 - 800`을 계산해서 `200`이라는 결과를 만든다.
- 3번 단계에서 두 스레드 모두 `balance = 200`을 대입한다.

- t1도 800원, t2도 800원을 출금했는데 잔액은 200원이 됨.
- 1,600원이 지출 되었는데 800원만 지출된 것 처럼 보이는 것.
- 실제로 이렇게 '동시에' 접근하는 문제가 발생하기도 하는것.

임계 영역

이런 문제가 발생한 근본 원인은 여러 쓰레드가 함께 사용하는 공유 자원을 여러 단계로 나누어 사용하기 때문.

1. 검증 단계 : 잔액(balance)이 출금액(amount)보다 많은지 확인한다.
2. 출금 단계 : 잔액(balance)을 출금액(amount)만큼 줄인다.

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

이 로직에는 하나의 큰 가정이 있다.

스레드 하나의 관점에서 `출금()`을 보면 1. 검증 단계에서 확인한 잔액(`balance`) 1000원은 2. 출금 단계에서 계산을 끝마칠 때 까지 같은 1000원으로 유지되어야 한다. 그래야 검증 단계에서 확인한 금액으로, 출금 단계에서 정확한 잔액을 계산할 수 있다.
그래야 검증 단계에서 확인한 1000원에 800원을 차감해서 200원이라는 잔액을 정확하게 계산할 수 있다.
결국 여기서는 내가 사용하는 값이 중간에 변경되지 않을 것이라는 가정이 있다.

그런데 만약 중간에 다른 스레드가 잔액의 값을 변경한다면, 큰 혼란이 발생한다. 1000원이라 생각한 잔액이 다른 값으로 변경되면 잔액이 전혀 다른 값으로 계산될 수 있다.

- **'내가 사용하는 값이 중간에 변경되지 않을 것'이라는 가정.**
- 만약 중간에 다른 쓰레드가 잔액의 값을 변경한다면 ?
    - 1,000원이라 생각한 잔액이 다른 값으로 변경된다면 잔액이 전혀 다른 값으로 계산될 수 있음.

공유 자원

잔액(balance)은 여러 쓰레드가 함께 사용하는 공유 자원.

따라서, 출금 로직을 수행하는 중간에 다른 쓰레드에서 이 값을 얼마든지 변경할 수 있음.

한 번에 하나의 쓰레드만 실행

만약 출금() 메서드를 한 번에 하나의 쓰레드만 실행할 수 있게 제한한다면 ?

Ex) t1, t2 쓰레드가 함께 출금()을 호출하면 t1 쓰레드가 먼저 메서드 완료 → t2 쓰레드가 메서드를 완료

- 이렇게 하면 공유 자원인 balance를 한 번에 하나의 쓰레드만 변경할 수 있게 됨.
- 따라서, 계산 중간에 다른 쓰레드가 balance의 값을 변경하는 부분을 걱정하지 않아도 된다는 것.

- 검증과 계산 두 단계는 **'한 번에 하나의 쓰레드만 실행해야함.'** 그래야 잔액이 중간에 변하지 않고 안전하게 계산할 수 있음.
- 근데 느려지지 않을까 ? 원래는 동시에 접근하고 완료되던 메서드가 순서대로 기다렸다가 접근하고 완료되어야 하는 거니까.

임계 영역 (Critical Section)

- 여러 쓰레드가 동시에 접근하면 데이터 불일치나 예상치 못한 동작이 발생할 수 있는 위험하고 또 중요한 코드 부분.
- 여러 쓰레드가 동시에 접근해서는 안되는 공유 자원을 접근하거나 수정하는 부분을 의미
    - Ex) 공유 변수, 공유 객체를 수정

즉, '출금()' 로직이 임계 영역인 것. = 잔액을 검증하는 시기 ~ 잔액의 계산을 완료하는 시기

> 둘 이상의 쓰레드가 동시에 접근하면 문제가 발생할 수 있는 '공유 자원에 대한 접근 코드 영역'.

> 반드시 한 번에 하나의 쓰레드만 실행되어야 하는 코드 블록.

즉,

- 공유 자원을 **읽고 쓰는 코드**
- 상태(state)를 변경하는 코드
- Race Condition이 발생할 수 있는 코드

```
count = count + 1;
// 겉보기에는 다음 코드가 '한 줄' 이지만, CPU와 JVM 입장에서는 아래 3단계로 나누어서 실행

1. 읽기(load)
   메모리(또는 CPU 캐시)에서 count 값을 읽어온다.

2. 계산(add)
   읽어온 값에 +1 계산을 한다.

3. 쓰기(store)
   계산된 값을 다시 메모리에 저장한다.
```

문제는 이 세 단계가 '끊어져 있음'
즉, 중간에 다른 쓰레드가 끼어들 수 있다는 것.
즉, 임계 영역이란 : 읽기 - 계산 - 쓰기가 끊어지지 않도록 한 번에 하나의 쓰레드만 실행하게 보호하는 코드 구역
synchronized는 읽기 + 계산 + 쓰기 -> 전체가 원자적으로 실행되어, 다른 쓰레드가 끼어들지 못함.

한 번에 하나의 쓰레드만 접근할 수 있도록 임계 영역을 보호할 수 있는 방법 ?

Synchronized 메서드

synchronized는 임계 영역(=공유 자원의 읽기–계산–쓰기 전체 과정)을 원자적으로 실행되도록 보장하여, 그 동안 다른 쓰레드가 진입하지 못하도록 막는 키워드이다.

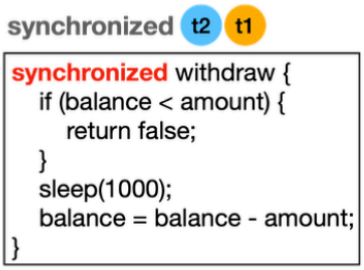
자바의 synchronized 키워드를 사용하면, 한 번에 하나의 쓰레드만 실행할 수 있는 코드 구간을 만들 수 있음.

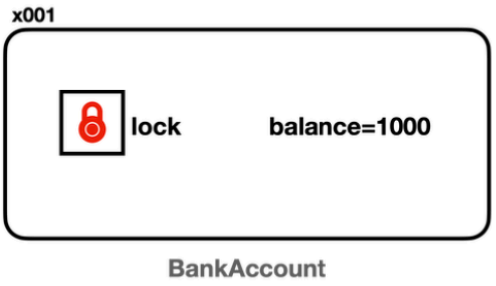
```
    @Override
synchronized public boolean withdraw(int amount) {
    log("거래 시작 : " + getClass().getSimpleName());

    // 잔고가 출금액 보다 적으면 진행하면 안됨.
    log("[검증 시작] 출금액 : " + amount + ", 잔액: " + balance);
    if(balance < amount) {
        log("[검증 실패]");
        return false;
    }

    // 잔고가 출금액 보다 많으면 진행
    log("[검증 완료] 출금액 : " + amount + ", 잔액: " + balance);
    sleep(1000); // 출금에 걸리는 시간으로 가정
    balance -= amount;

    log("[출금 완료] 출금액 : " + amount + ", 잔액: " + balance);

    log("거래 종료");
    return true;
```

```
    }

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

```
22:38:45.791 [      t2] 거래 시작 : BankAccountV2
22:38:45.798 [      t2] [검증 시작] 출금액 : 800, 잔액: 1000
22:38:45.798 [      t2] [검증 완료] 출금액 : 800, 잔액: 1000
22:38:46.270 [    main] t1 state: BLOCKED
22:38:46.270 [    main] t2 state: TIMED_WAITING
22:38:46.802 [      t2] [출금 완료] 출금액 : 800, 잔액: 200
22:38:46.803 [      t2] 거래 종료
22:38:46.803 [      t1] 거래 시작 : BankAccountV2
22:38:46.803 [      t1] [검증 시작] 출금액 : 800, 잔액: 200
22:38:46.803 [      t1] [검증 실패]
22:38:46.804 [    main] 최종 잔액 : 200
```

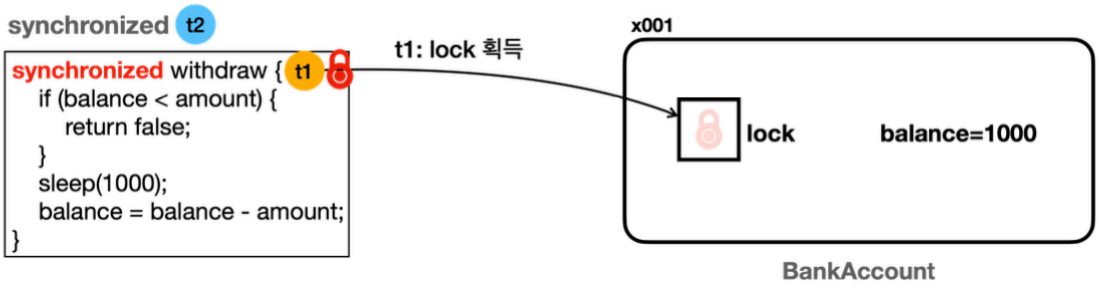
synchronized 분석



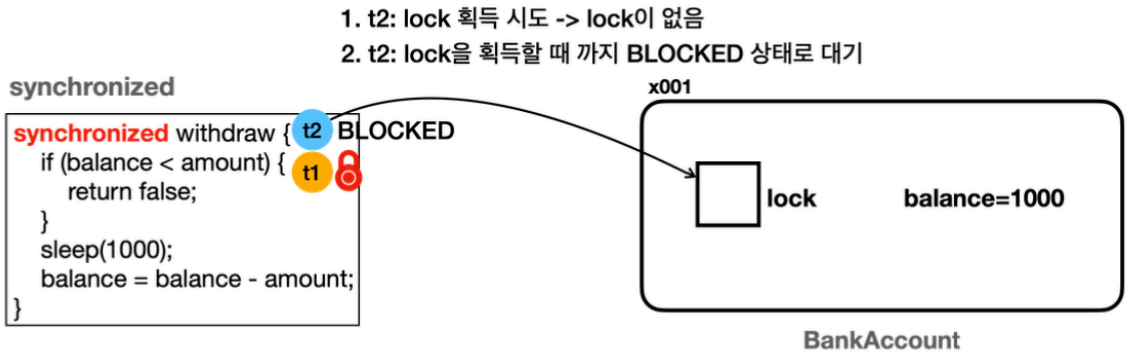


- **모든 객체(인스턴스)는 내부에 자신만의 락(Lock)을 가지고 있음.**
  - 모니터 락 이라고 부르며, 객체 내부에 존재
  - 모니터 락(Monitor Lock)의 **주 목적**은 바로 **동시성 문제 해결**이에요.
  - 즉, 여러 쓰레드가 동시에 접근하면 데이터가 깨지거나 예측 불가능한 동작이 발생할 수 있는 **임계 영역(Critical Section)**을 보호하기 위해 존재합니다.
- 쓰레드가 synchronized 키워드가 있는 메서드에 진입하려면 반드시 해당 인스턴스의 락이 있어야 함.

- 여기서는 BankAccount(x001) 인스턴스의 synchronized withdraw() 메서드를 호출하므로 이 인스턴스의 락이 필요함.

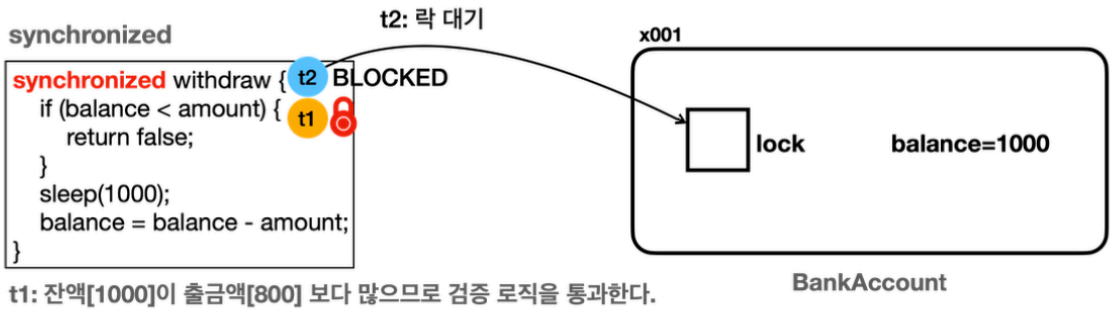


- 쓰레드 t1이 먼저 synchronized 키워드가 있는 withdraw() 메서드를 호출.
- synchronized 메서드를 호출하려면 먼저 해당 인스턴스의 락이 필요함.
- 락이 있으므로 쓰레드 t1은 BankAccount(x001) 인스턴스에 있는 락을 획득.



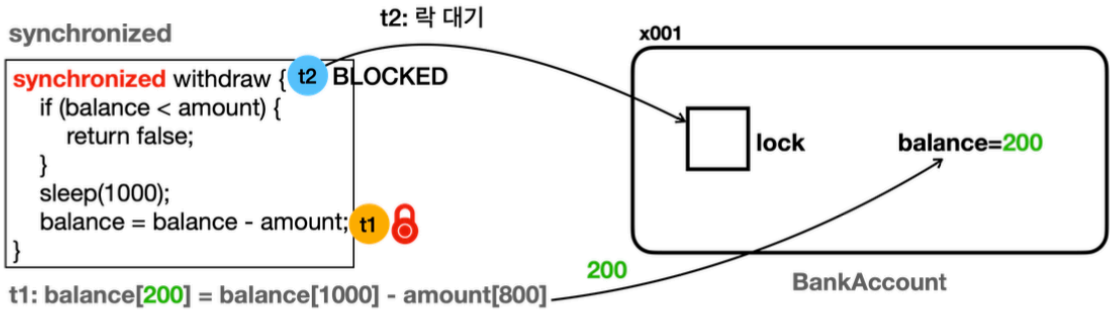
- 쓰레드 t1은 해당 인스턴스의 락을 획득했기 때문에 withdraw() 메서드에 진입.
- 쓰레드 t2도 withdraw() 메서드 호출을 시도하나, synchronized 메서드를 호출하려면 먼저 해당 인스턴스의 락이 필요.
- 쓰레드 t2sms BankAccount(x001) 인스턴스에 있는 락 획득을 시도하나, 락이 없음. 이렇게 락이 없으면 t2는 'BLOCKED' 상태로 대기
  - t2 쓰레드의 상태는 RUNNABLE → BLOCKED 상태로 변하고, 락을 획득할 때 까지 무한정 대기.

> 참고로 BLOCKED 상태가 되면 락을 획득하기 전 까지는 계속 대기하며 CPU의 실행 스케줄링에 들어가지 않음.



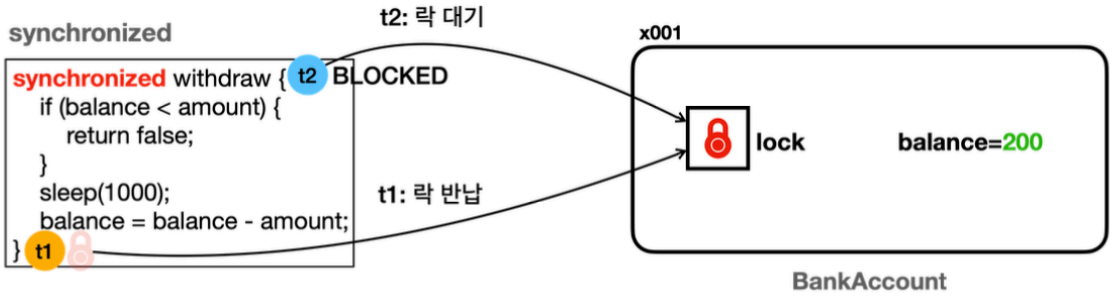
t1: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.

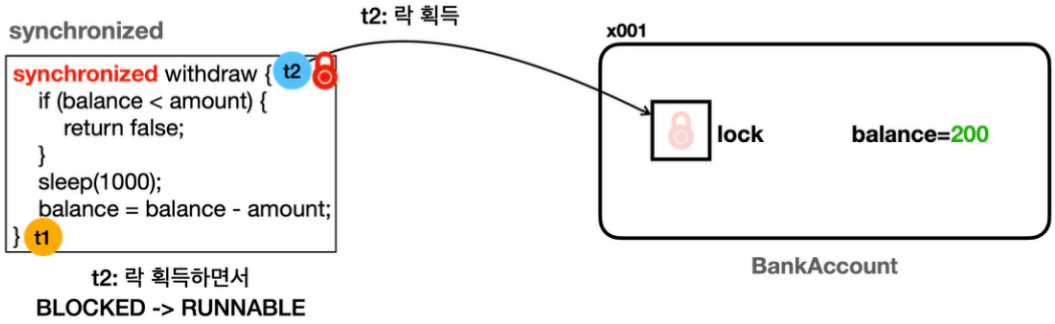
- `t1` : 출금을 위한 검증 로직을 수행한다. 조건을 만족하므로 검증 로직을 통과한다.
  - 잔액[1000]이 출금액[800] 보다 많으므로 통과한다.



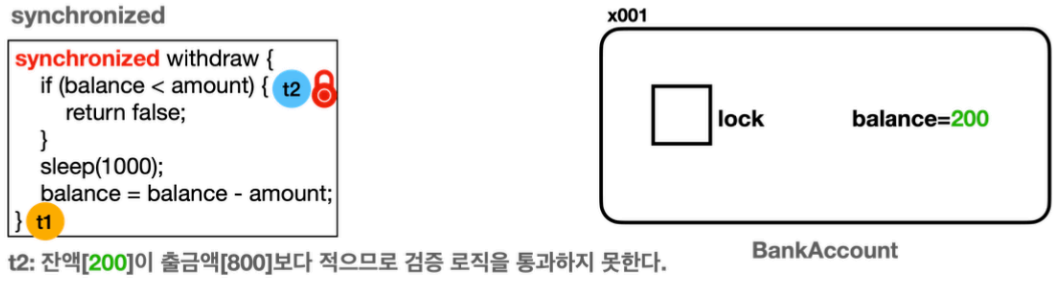
t1: balance[200] = balance[1000] - amount[800]

- `t1` : 잔액 1000원에서 800원을 출금하고 계산 결과인 200원을 잔액(`balance`)에 반영한다.

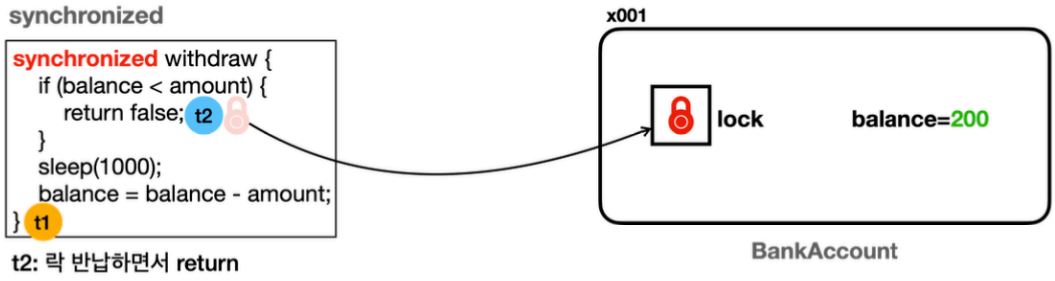




- `t2` : 인스턴스에 락이 반납되면 **락 획득을 대기하는 스레드는 자동으로 락을 획득**한다.
  - 이때 락을 획득한 스레드는 `BLOCKED` ➔ `RUNNABLE` 상태가 되고, 다시 코드를 실행한다.



- 스레드 `t2` 는 해당 인스턴스의 락을 획득했기 때문에 `withdraw()` 메서드에 진입할 수 있다.
- `t2` : 출금을 위한 검증 로직을 수행한다. 조건을 만족하지 않으므로 `false` 를 반환한다.
  - 이때 잔액(`balance`)은 200원이다. 800원을 출금해야 하므로 조건을 만족하지 않는다.



t1: 800원 출금 완료
t2: 잔액 부족으로 출금 실패
원금 1000원, 최종 잔액 200원

- `t2` : 락을 반납하면서 `return` 한다.

락을 획득하는 순서는 보장되지 않는다.

만약, BankAccount(x001)에 withdraw() 메서드를 여러 쓰레드에서 동시에 호출한다면, 1개의 쓰레드만 락을 획득하고 나머지는 모두 'BLOCKED'상태가 됨.

이 때 어떤 순서로 락을 획득하는지는 자바 표준에 정의되어 있지 않음. 따라서, 순서를 보장하지 않고 환경에 따라 순서가 달라질 수 있음.

> **volatile 키워드를 사용하지 않아도, synchronized 안에서 접근하는 변수의 메모리 가시성 문제는 해결 됨.**
> **가시성(Visibility)**
>
> → 락을 획득할 때 메인 메모리에서 최신값을 읽어오고
>
> 락을 풀 때 변경된 값을 메인 메모리에 반영.

synchronized 코드 블럭

synchronized의 가장 큰 장점이자 단점은 **한 번에 하나의 쓰레드만 실행할 수 있다는 점임.**

여러 스레드가 동시에 실행하지 못하기 때문에, 전체로 보면 성능이 떨어질 수 있다. 따라서 `synchronized` 를 통해 여러 스레드를 동시에 실행할 수 없는 코드 구간은 꼭! 필요한 곳으로 한정해서 설정해야 함.

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

        // 잔고가 출금액 보다 적으면 진행하면 안됨.
        log("[검증 시작] 출금액 : " + amount + ", 잔액: " + balance);
        if(balance < amount) {
            log("[검증 실패]");
            return false;
        }

        // 잔고가 출금액 보다 많으면 진행
        log("[검증 완료] 출금액 : " + amount + ", 잔액: " + balance);
        sleep(1000); // 출금에 걸리는 시간으로 가정
        balance -= amount;

        log("[출금 완료] 출금액 : " + amount + ", 잔액: " + balance);

        log("거래 종료");
        return true;
    }
```

- 처음에 로그를 출력하는 '거래 시작', 그리고 마지막에 로그를 출력하는 '거래 종료' 부분은 공유 자원을 '**전혀 사용하지 않음**'
- 이런 부분은 동시에 실행해도 아무 문제가 발생하지 않음.

따라서 임계 영역은 다음과 같음.

```
public synchronized boolean withdraw(int amount) {
    log("거래 시작: " + getClass().getSimpleName());
    // ==임계 영역 시작==
    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);
    // ==임계 영역 종료==

    log("거래 종료");
  return true;
}
```

- 하지만 메서드에 synchronized 설정 시 임계 영역은 메서드 전체가 됨.
- 이런 문제를 해결하기 위해 자바에서는 synchronized를 메서드 전체가 아닌 특정 코드 블록에 적용할 수 있는 기능을 제공.

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

        // 잔고가 출금액 보다 적으면 진행하면 안됨.
        log("[검증 시작] 출금액 : " + amount + ", 잔액: " + balance);

        synchronized (this) {
            if (balance < amount) {
                log("[검증 실패]");
                return false;
            }

            // 잔고가 출금액 보다 많으면 진행
```

```
            log("[검증 완료] 출금액 : " + amount + ", 잔액: " + balance);
            sleep(1000); // 출금에 걸리는 시간으로 가정
            balance -= amount;

            log("[출금 완료] 출금액 : " + amount + ", 잔액: " + balance);
        }

        log("거래 종료");
        return true;
    }
```

- synchronized(this) {} : 안전한 임계 영역을 코드 블럭으로 지정
- synchronized(this) : this는 락을 획득할 인스턴스의 참조

```
22:57:11.898 [      t2] 거래 시작 : BankAccountV3
22:57:11.898 [      t1] 거래 시작 : BankAccountV3
22:57:11.903 [      t1] [검증 시작] 출금액 : 800, 잔액: 1000
22:57:11.903 [      t2] [검증 시작] 출금액 : 800, 잔액: 1000
22:57:11.903 [      t1] [검증 완료] 출금액 : 800, 잔액: 1000
22:57:12.382 [    main] t1 state: TIMED_WAITING
22:57:12.382 [    main] t2 state: BLOCKED
22:57:12.908 [      t1] [출금 완료] 출금액 : 800, 잔액: 200
22:57:12.908 [      t1] 거래 종료
22:57:12.908 [      t2] [검증 실패]
22:57:12.910 [    main] 최종 잔액 : 200
```

- 하나의 쓰레드만 실행할 수 있는 안전한 임계 영역은 '**가능한 최소한의 범위에 적용해야 함.**'
- 그래야, 도시에 여러 쓰레드가 실행할 수 있는 부분을 늘려서, 전체적인 처리 성능을 더 높일 수 있음.

synchronized 동기화 정리

자바에서 동기화(synchronization)는 여러 스레드가 동시에 접근할 수 있는 자원(예: 객체, 메서드)에 대해 일관성 있고 안전한 접근을 보장하기 위한 메커니즘이다. 동기화는 주로 멀티스레드 환경에서 발생할 수 있는 문제, 예를 들어 데 이터 손상이나 예기치 않은 결과를 방지하기 위해 사용된다.

메서드 동기화 : 메서드에 접근하는 쓰레드가 하나뿐이도록 보장

```
public synchronized void synchronizedMethod() {
    // 코드
}
```

블록 동기화 : 코드 블록을 synchronized로 감싸서 동기화 구현

```
public void method() {
  synchronized(this) {
        // 동기화된 코드
    }
}
```

이런 동기화를 사용하면 다음 문제들을 해결할 수 있다.

- **경합 조건(Race condition)**: 두 개 이상의 스레드가 경쟁적으로 동일한 자원을 수정할 때 발생하는 문제.
- **데이터 일관성**: 여러 스레드가 동시에 읽고 쓰는 데이터의 일관성을 유지.

동기화는 멀티스레드 환경에서 필수적인 기능이지만, 과도하게 사용할 경우 성능 저하를 초래할 수 있으므로 꼭 필요한곳에 적절히 사용해야 한다.

> synchronized가 붙지 않은 모든 코드 영역은 락을 획득하려는 시도조차 하지 않는다. 락을 사용하고, 대기하고, 반환하는 모든 메커니즘은 오직 synchronized가 붙은 블록/메서드에서만 작동한다.

❌ synchronized가 없는 코드에서는?

- 락 획득 없음
- 락 반환 없음

🔥 중요한 의미

즉, 모든 Java 객체는 “모니터 락”을 내부적으로 가지고 있지만...

→ **그 락을 사용할지 말지는 오직 synchronized가 결정한다.**

→ synchronized가 없는 메서드는 그 객체의 락을 전혀 건드리지 않는다.

→ 심지어 객체 사이에 충돌도 없다.

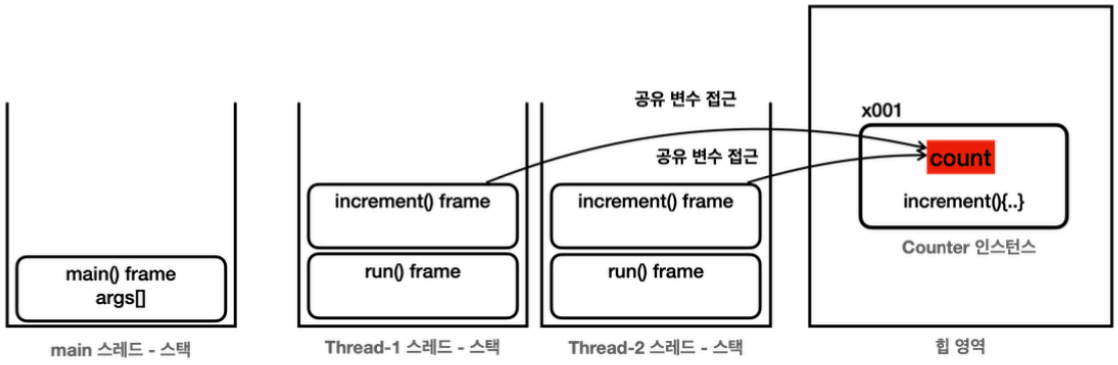
문제와 풀이

문제 1.

올바른 결과 : 20,000이 나오기 위한 코드 수정

```
package thread.test;

public class SyncTest1BadMain {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    counter.increment();
                }
            }
        };
        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println("결과: " + counter.getCount());
    }

    static class Counter {
        private int count = 0;

        public synchronized void increment() {
            count = count + 1;
        }

        public int getCount() {
            return count;
        }
```

```
    }
}
```



여기서 공유 자원을 사용하는 `increment()` 메서드는 다음과 같이 3단계로 나누어져 있다.

`count = count + 1`

1. `count`의 값을 읽는다.
2. 읽은 `count`의 값에 1을 더한다.
3. 더한 결과를 `count`에 다시 저장한다.

- increment가 값을 변경하는 세 가지 단계
  - read, compute, write

> read, compute 단계 즉, write가 되기 전 단계에서 다른 쓰레드가 값을 읽으려할 때 아직 wrtie되지 않았으므로 그 전의 값을 읽어감.
>
> 읽기, 계산, 쓰기라는 여러 단계를 '원자적'으로 묶지 않아서 서로의 중간 단계 사이로 끼어들 수 있기 때문에 일어나는 것.
>
> 결국 이게 동시성 문제의 본질.

단일 스레드가 공유 자원에 접근하는 상황

`count` 값이 0이라고 가정하겠다.

1. `count`의 값을 읽는다. `count` 값은 0이다.
2. 읽은 `count`의 값에 1을 더한다. 0 + 1 = 1이다.
3. 더한 결과를 `count`에 다시 저장한다. `count` 값은 1이다.

이처럼 단일 스레드가 공유 자원에 접근하는 경우는 아무런 문제가 없다.

여러 스레드가 공유 자원에 함께 접근하는 상황

`count = count + 1`

`count` 값이 0이라고 가정하겠다. 2개의 스레드가 동시에 `increment()` 메서드를 호출한다.

- **스레드1**: `count`의 값을 읽는다. `count` 값은 0이다.
- **스레드2**: `count`의 값을 읽는다. `count` 값은 0이다.

- **스레드1**: 읽은 `count`의 값에 1을 더한다. 0 + 1 = 1이다.
- **스레드2**: 읽은 `count`의 값에 1을 더한다. 0 + 1 = 1이다.

- **스레드1**: 더한 결과를 `count`에 다시 저장한다. `count` 값은 1이다.
- **스레드2**: 더한 결과를 `count`에 다시 저장한다. `count` 값은 1이다.

스레드 2개가 `increment()`를 호출하기 때문에 기대하는 `count`의 결과는 2가 되어야 한다.
하지만 둘이 동시에 실행되기 때문에, 처음에 둘다 `count`의 값을 0으로 읽었다.
여기서 잘 보면 `count`의 값을 읽어서 계산하는 부분과 그 결과를 `count`에 다시 넣는 부분으로 나누어져 있다.
따라서 여러 스레드가 동시에 실행되면 지금과 같은 문제가 발생할 수 있다.
따라서 `synchronized` 키워드를 사용해서 한 번에 하나의 스레드만 실행할 수 있도록, 안전한 임계 영역을 만들어야 한다.

문제 2. 지역 변수의 공유

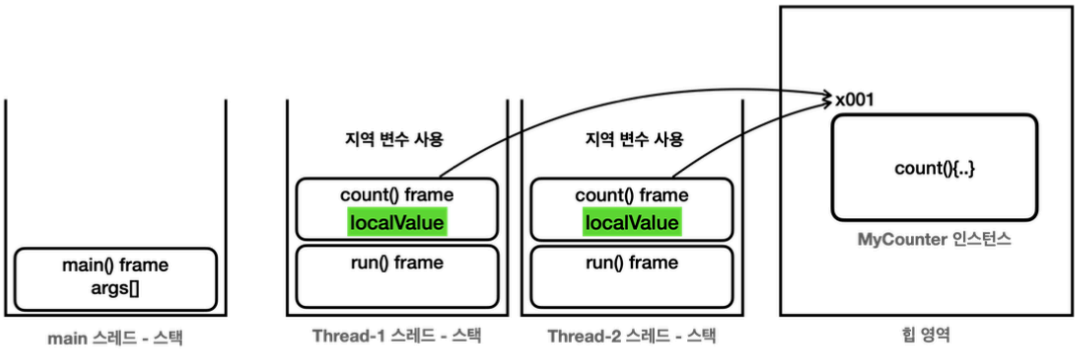
다음 코드에서 `MyTask` 의 `run()` 메서드는 두 스레드에서 동시에 실행한다.

다음 코드의 실행 결과를 예측해보자.
그리고 `localValue` 지역 변수에 동시성 문제가 발생하는지 하지 않는지 생각해보자.

```
package thread.test;

import static util.MyLoggerThread.log;

public class SyncTest2Main {
    public static void main(String[] args) throws InterruptedException {
        MyCounter myCounter = new MyCounter();
```

```
        Runnable task = new Runnable() {
            @Override
            public void run() {
                myCounter.count();
            } };
        Thread thread1 = new Thread(task, "Thread-1");
        Thread thread2 = new Thread(task, "Thread-2");
        thread1.start();
        thread2.start();
    }
    static class MyCounter {
        public void count() {
            int localValue = 0;
            for (int i = 0; i < 1000; i++) {
                localValue = localValue + 1;
            }
            log("결과: " + localValue); }
    }
}
```

- 이 코드는 동시성 문제가 발생하지 않음.
- 왜 ? → 이 결과는 localValue라는 지역 변수에 대한 값을 호출하고 있음.
- 메서드 내에 지역변수는 쓰레드마다 독립적으로 값을 갖고있으므로, 동시성 문제가 발생하지 않음.
    - 동시성 문제는 '공유 자원'에 수정이 발생할 때 일어나는데, 지역 변수는 쓰레드 독립적이기 때문.



- `localValue`는 지역 변수이다.
- 스택 영역은 각각의 스레드가 가지는 별도의 메모리 공간이다. 이 메모리 공간은 다른 스레드와 공유하지 않는다.
- 지역 변수는 스레드의 개별 저장 공간인 스택 영역에 생성된다.
- 따라서 **지역 변수는 절대로! 다른 스레드와 공유되지 않는다!**
- 이런 이유로 지역 변수는 동기화에 대한 걱정을 하지 않아도 된다.
  - 여기에 `synchronized`를 사용하면 아무 이득도 얻을 수 없다. 성능만 느려진다!
- 지역 변수를 제외한, 인스턴스의 멤버 변수(필드), 클래스 변수 등은 공유될 수 있다.

- **지역 변수는 ' 절대로' 다른 쓰레드와 '공유되지 않음'**
- 때문에, 동기화 문제를 걱정하지 않아도 되며, 지역 변수 내에서만 사용되는 값을 위해 synchronized를 사용할 필요도 없으며, 사용해도 성능만 나빠짐.

문제 3. final 필드

다음에서 `value` 필드(멤버 변수)는 공유되는 값이다. 멀티스레드 상황에서 문제가 될 수 있을까?

```
class Immutable {
    private final int value;
    public Immutable(int value) {
        this.value = value;
}
    public int getValue() {
        return value;
} }
```

- 문제가 되지 않음.
- '공유 자원'을 '수정'할 때 문제가 되는 것이지, 읽는 것은 문제가 안됨.

여러 스레드가 공유 자원에 접근하는 것 자체는 사실 문제가 되지 않는다. 진짜 문제는 공유 자원을 사용하는 중간에 다른 스레드가 공유 자원의 값을 변경해버리기 때문에 발생한다. 결

국 변경이 문제가 되는 것이다.
여러 스레드가 접근 가능한 공유 자원이라도 그 값을 아무도 변경할 수 없다면 문제 되지 않는다. 이 경우 모든 스레드가항상 같은 값을 읽기 때문이다.
필드에 `final` 이 붙으면 어떤 스레드도 값을 변경할 수 없다. 따라서 멀티스레드 상황에 문제 없는 안전한 공유 자원이 된다.

정리

자바는 처음부터 멀티스레드를 고려하고 나온 언어이다. 그래서 자바 1.0 부터 `synchronized` 같은 동기화 방법을프로그래밍 언어의 문법에 포함해서 제공한다.

synchronized 장점

- 프로그래밍 언어에 문법으로 제공
- 아주 편리한 사용
- ****자동 잠금 해제***: `synchronized` 메서드나 블록이 완료되면 자동으로 락을 대기중인 다른 스레드의 잠금이 해제된다. 개발자가 직접 특정 스레드를 깨우도록 관리해야 한다면, 매우 어렵고 번거로울 것이다.

`synchronized` 는 매우 편리하지만, 제공하는 기능이 너무 단순하다는 단점이 있다. 시간이 점점 지나면서 멀티스레드가 더 중요해지고 점점 더 복잡한 동시성 개발 방법들이 필요해졌다.

synchronized 단점

- ****무한 대기***: `BLOCKED` 상태의 스레드는 락이 풀릴 때 까지 무한 대기한다.
  - 특정 시간까지만 대기하는 타임아웃X
  - 중간에 인터럽트X
- ****공정성****: 락이 돌아왔을 때 `BLOCKED` 상태의 여러 스레드 중에 어떤 스레드가 락을 획득할 지 알 수 없다. 최악의경우 특정 스레드가 너무 오랜기간 락을 획득하지 못할 수 있다.

`synchronized` 의 가장 치명적인 단점은 락을 얻기 위해 `BLOCKED` 상태가 되면 락을 얻을 때까지 무한 대기한다는점이다. 비유를 하자면 맛집에 한 번 줄을 서면 10시간이든 100시간이든 밥을 먹을 때까지 강제적으로 계속 기다려야한다는 점이다.

예를 들어 웹 애플리케이션의 경우 고객이 어떤 요청을 했는데, 화면에 계속 요청 중만 뜨고, 응답을 못 받는 것이다. 차라리 너무 오랜 시간이 지나면, 시스템에 사용자가 너무 많아서 다음에 다시 시도해달라고 하는 식의 응답을 주는 것이더 나은 선택일 것이다.

결국 더 유연하고, 더 세밀한 제어가 가능한 방법들이 필요하게 되었다. 이런 문제를 해결하기 위해 자바 1.5부터
`java.util.concurrent` 라는 동시성 문제 해결을 위한 패키지가 추가된다.

참고로 단순하고 편리하게 사용하기에는 `synchronized` 가 좋으므로, 목적에 부합한다면 `synchronized` 를 사용하면 된다.