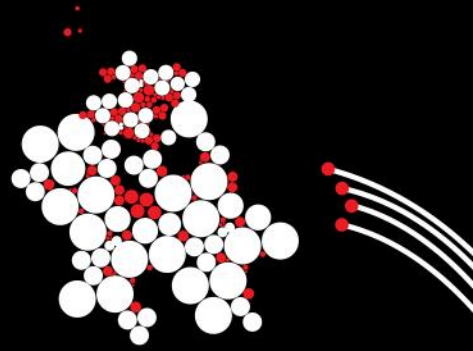
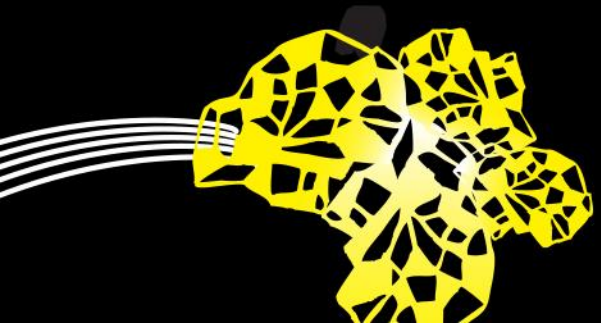
UNIVERSITY OF TWENTE.



Synchronisation in Java

Topic of Software Systems (TCS module 2)

Lecturer: Marieke Huisman



MUTUAL EXCLUSION IN JAVA

- The `Lock` interface in Java
  - Method `lock` to acquire the lock
  - Method `unlock` to release the lock
  - The `lock` method blocks until the lock becomes available
  - Method `tryLock()` either acquires the lock and returns true, or returns false
  - Popular implementation: `java.util.concurrent.locks.ReentrantLock`

- Java offers an easier method: the **`synchronized`** keyword

```
synchronized (someObject) {
    // critical section
}
```

- In Java, every object is also a lock with the synchronized keyword

SYNCHRONIZED BLOCKS

Every object is a lock with the **`synchronized`** keyword

```
synchronized (someObject) {
    // critical section
}
```

`Thread t` arrives at synchronized code block for `someObject`

- If no other thread holds the lock on `someObject`, this thread will get the lock
- If another thread already has the lock, this thread will start waiting for the lock

`Thread t` leaves synchronized code block for `someObject`

- Lock of `someObject` is passed to an arbitrary thread waiting for it

There is no guarantee your thread will EVER get the lock!

Beware: if you synchronize on different objects, you might get a different result than you think!

SYNCHRONIZE THE CALL

```
public class Deposit100ATM implements Runnable {
    private BankAccount acc;
    public Deposit100ATM(BankAccount acc) { this.acc = acc; }
    public void run() {
        synchronized (acc) {
            acc.deposit(100.0);
        }
    }
}
```

This blocks the threads from executing these lines in parallel.

Client handles the synchronisation problem

SYNCHRONIZE THE CALL

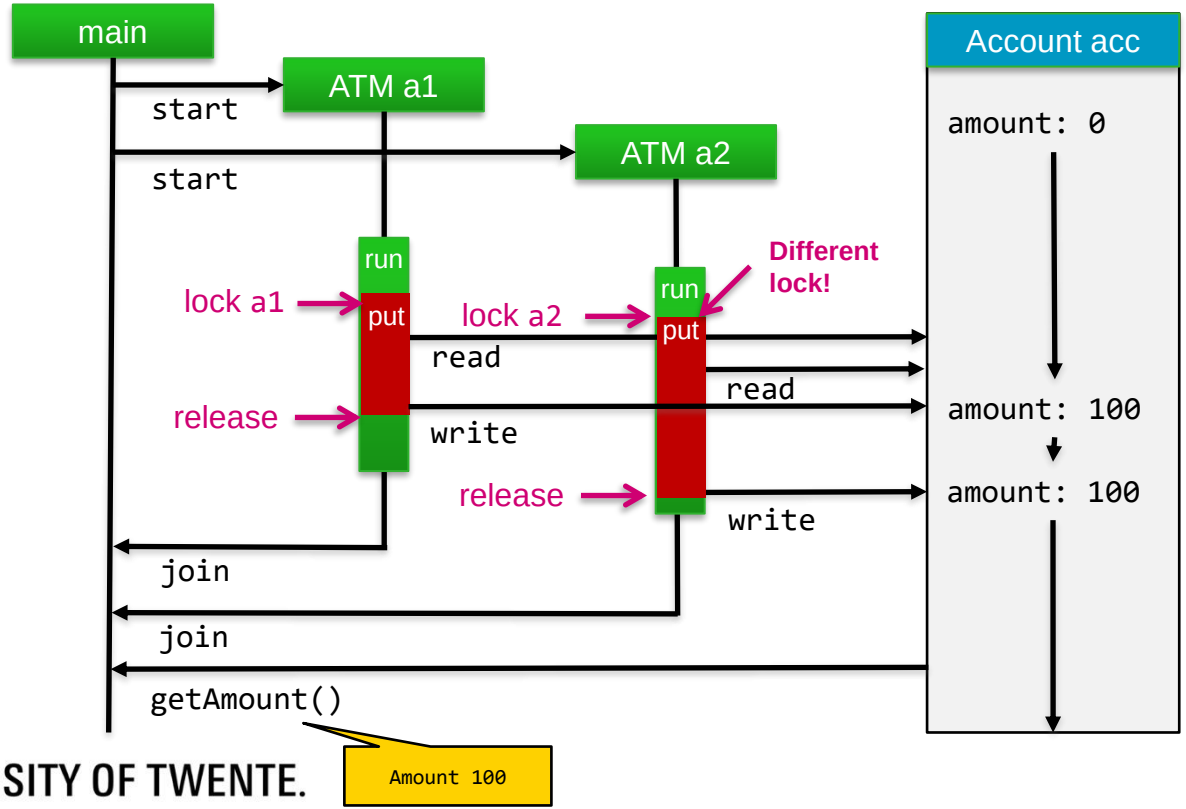
```
public class Deposit100ATM implements Runnable {
    private BankAccount acc;
    public Deposit100ATM(BankAccount acc) { this.acc = acc; }
    public void run() {
        synchronized (this) {
            acc.deposit(100.0);
        }
    }
}
```

This will NOT work!

Threads will use different locks!

They have to use the same lock

A POSSIBLE EXECUTION



SYNCHRONIZE THE CALL

```
public class Deposit100ATM implements Runnable {
    private BankAccount acc;
    public Deposit100ATM(BankAccount acc) { this.acc = acc; }
    public void run() {
        synchronized (this) {
            acc.deposit(100.0);
        }
    }
}
```

This will NOT work!

Threads will use different locks!

They have to use the same lock

There is a risk in relying on the client to lock correctly.

SYNCHRONISATION BY SERVER

```
public class BankAccount {
    …
    public void deposit(double val) {
        synchronized (this) {
            amount = amount + val;
        }
    }
    …
}
```

- Advantage: client does not have to care about synchronization
- Server knows all calls are correctly synchronized

Alternative:

```
public synchronized void deposit(double val) {
    amount = amount + val;
}
```

ALTERNATIVE: LOCK INTERFACE

Using the `ReentrantLock`:

```
public class BankAccount {
    Lock l = new ReentrantLock();

    ...

    public void deposit(double val) {
        l.lock();
        amount = amount + val;
        l.unlock();
    }
}
```

- More flexilibility
- Lock/unlock different methods
- Different lock implementations
- Easier to forget to unlock