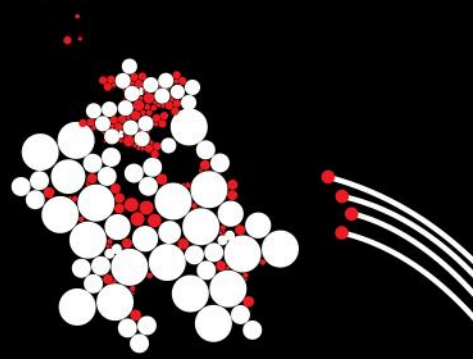
UNIVERSITY OF TWENTE.

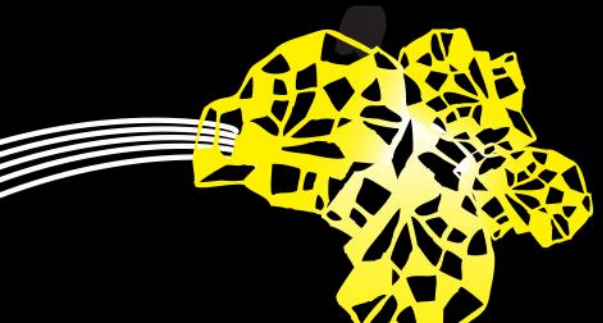


P6.1: CONCURRENCY

TOM VAN DIJK

201700117-1B MODULE 2: SOFTWARE SYSTEMS

16 DECEMBER 2019



PROGRAMMING LINE OVERVIEW

Week 1 Values and variables Control flow	**Week 2** Classes and objects Testing	**Week 3** Interfaces and Inheritance Subtyping Security 1
Week 4 Arrays and Lists List implementations Collections	**Week 5** Exceptions I/O Streams and MVC Security 2	**Week 6** Concurrency Project kick-off IDE Tips & Tricks
Week 7 Basic Networking Networking and Multithreading GUIs	**Week 8/9** Advanced Java facilities Test	**Week 10** Project Test resit

CONTENTS

- Running multiple threads in parallel
- Synchronization (“mutual exclusion”)
- Communication between threads

- Literature:
  - Core Java, Chapter 14, until *BlockingQueues* (p. 819 - 877)
  - Eck,12.1 – 12.3

WHAT IS CONCURRENCY?

- Most of your programs so far: single-threaded
  - Only one execution of the code
  - Also: only one execution to think about

- Now: multi-threaded programs (also called concurrent or parallel)
  - As if multiple programs are running in the same “Java world”
  - Also: more difficult to reason about code correctness
    - Intentional *and* unintentional interaction between threads

- Actually: JVM already has extra threads, for example for garbage collection!

WHAT IS CONCURRENCY?

- Why? More efficient!
  - Computers have multiple processors (and processors have multiple cores)
    - Great for divide-and-conquer algorithms like mergesort!
  - Threads can be blocked
    - Waiting for disk or network I/O
    - Graphical user interfaces

- How do threads communicate?
  - Via shared memory and/or message passing and/or monitors

WHAT IS CONCURRENCY?

- Thinking about your multi-threaded program

```
// A little experiment with a naive implementation of mergesort
// using the advanced Java work stealing thread pool to
// sort an ArrayList of random Integes

Available hardware threads: 6
to sort 10000000 integers with Java's default TimSort: 5.455 sec.
to sort 10000000 integers with parallel mergesort 1: 10.928 sec.
to sort 10000000 integers with parallel mergesort 2: 6.585 sec.
to sort 10000000 integers with parallel mergesort 4: 4.646 sec.
to sort 10000000 integers with parallel mergesort 8: 4.315 sec.
```

- Multiple “workers” (threads) collaborate on a result
- Mergesort: let other threads sort the smaller array concurrently, then merge sequentially
- User interface: some threads handle input, other threads handle output

CHALLENGES WITH CONCURRENCY

- Thinking about your multi-threaded program is hard
  - Especially if you really want your program to be fast and efficient
  - Concurrency bugs are hard to debug: Heissenbugs
- Two examples:
  - Accessing objects from multiple threads: data race / race condition
  - Multiple threads waiting for each other: deadlock
- Concurrency bugs can be deadly
  - Famous “Therac-25” race condition
  - North American Blackout of 2003

CONCURRENCY IN JAVA

- Built-in support for threads (creating, waiting for them to finish, etc)
- Support for object-based synchronization (controlling access to shared resources)
- Object-based wait-notify mechanism (wait for a thread to signal)
- Extensive library for concurrency: java.util.concurrent

CREATING A THREAD

- As a subclass of Thread
  - Define a subclass of class Thread
  - Override the inherited method run
  - Construct an object of your subclass
  - Call the method start to start the thread!

- Better: implement a Runnable
  - Define class that implements the interface Runnable
  - Implement the method run
  - Construct an object of your class
  - Construct a new Thread(yourRunnableObject) and start() it!

EXAMPLE

- What will happen if you run this?

```
public class SaySomething implements Runnable {
    private String text;
    public SaySomething(String text) {
        this.text = text;
    }
    public void run() {
        for (int i=0; i<1000; i++) System.out.println(text);
    }
}

Thread threadOne = new Thread(new SaySomething(“Hello from thread 1!”));
Thread threadTwo = new Thread(new SaySomething(“Hello from thread 2!”));
threadOne.start();
threadTwo.start();
```

EXAMPLE

- What will happen if you run this?

```
public class SaySomething implements Runnable {
    private String text;
    public SaySomething(String text) {
        this.text = text;
    }
    public void run() {
        for (int i=0; i<1000; i++) System.out.println(text);
    }
}

for (int i=0; i<1000; i++) {
    new Thread(new SaySomething(“Hello from thread “ + i + !”)).start();
}
```

THREADING FUNCTIONALITY

- After creating a Thread t, you can do things with the thread
  - Call `t.start()` to start the thread
  - Call `t.join()` to wait until the thread is terminated
  - Call `t.join(n)` to wait n milliseconds or until the thread is terminated
  - Call `t.interrupt()` to cause an `InterruptedException` to be thrown in the running thread
  - Use `setName(name)` and `getName()` to name your thread and retrieve its name
  - Call `t.setDaemon(true)` before starting the thread to make it a daemon thread
    - If all non-daemon threads are terminated, the program terminates
    - Use daemon threads for background supporting tasks

- Methods `suspend`, `resume`, `stop` are deprecated
  - They often lead to deadlocks, so it is not a good practice to use them

EXAMPLE

- What will happen if you run this?

```
Thread threadOne = new Thread(new SaySomething(“Hello from thread 1!”));
Thread threadTwo = new Thread(new SaySomething(“Hello from thread 2!”));
threadOne.start();
threadTwo.start();

try {
    threadOne.join();
} catch (InterruptedException ex) {
    // oops, the main thread was interrupted!
    ex.printStackTrace();
} finally {
    System.exit(0);
}
```

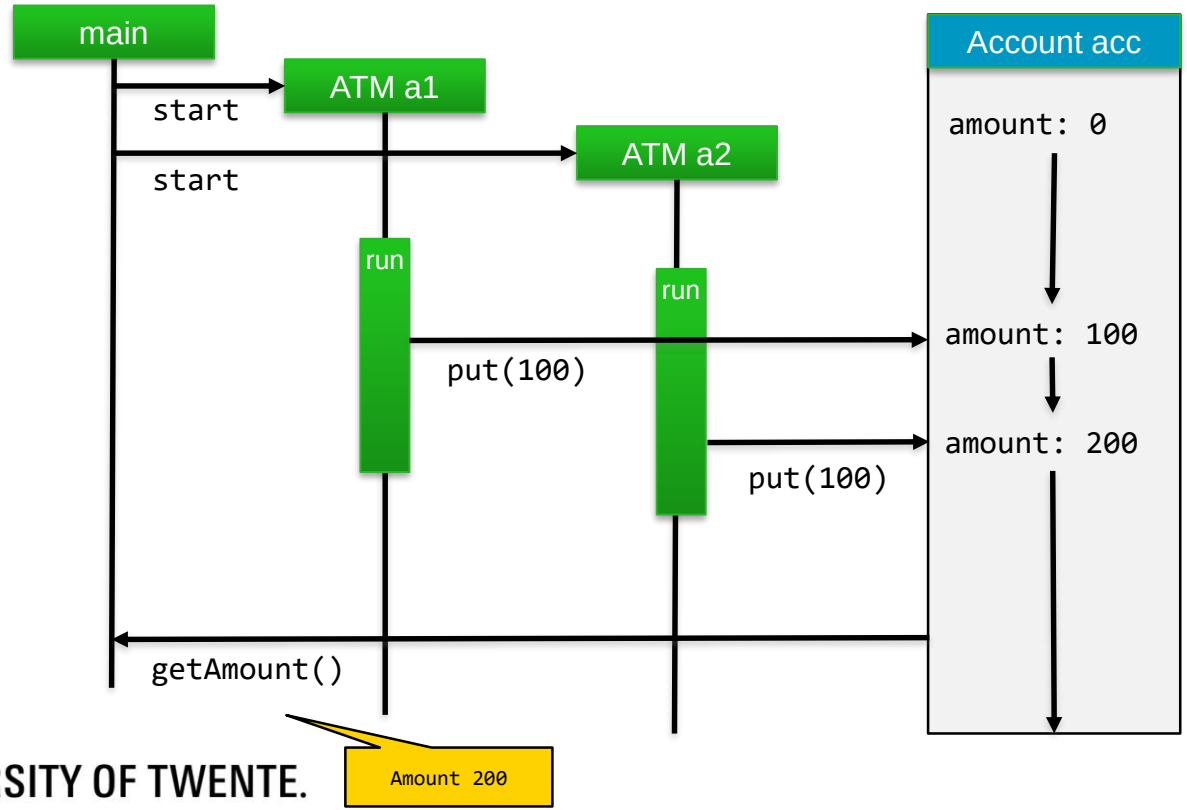
EXAMPLE: BANK ACCOUNT AND ATM

```
public class BankAccount {
    private double amount = 0.0;
    public void deposit(double val) { amount = amount + val; }
    public void withdraw(double val) { amount = amount - val; }
    public double getAmount() { return amount; }
}
```

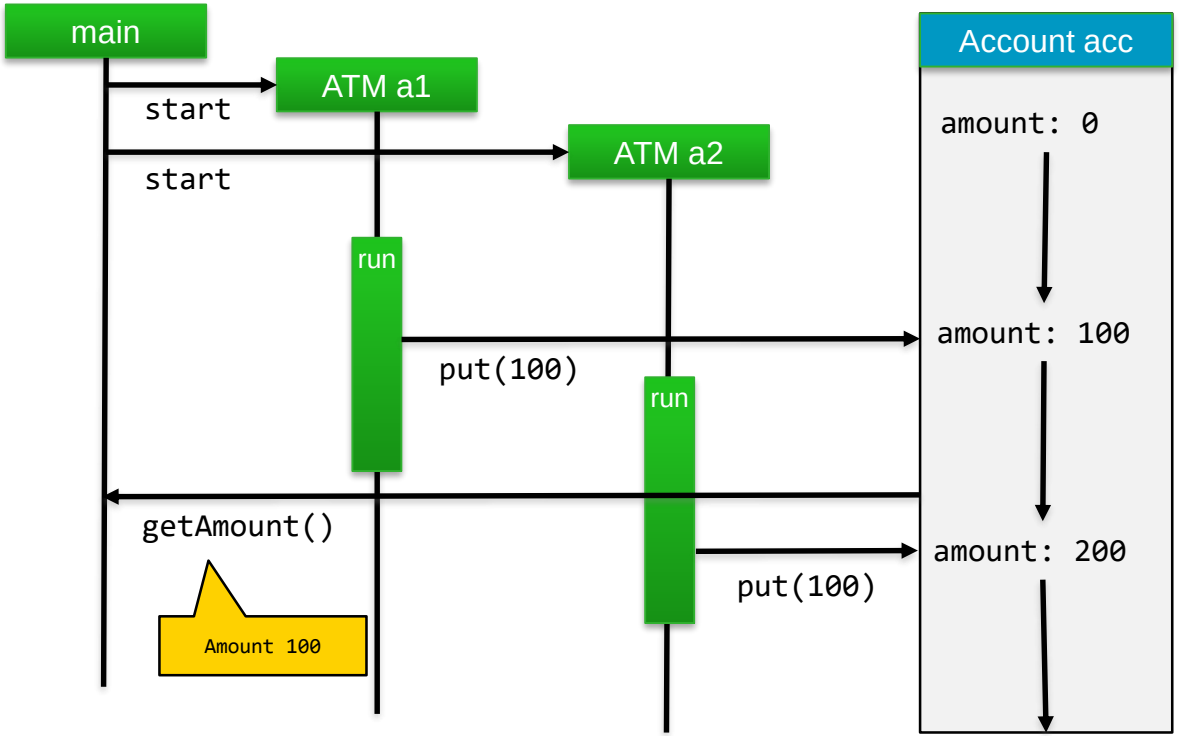
WHAT WILL THIS PROGRAM PRINT?

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

public static void main(String[] args) {
    BankAccount acc = new BankAccount();
    Deposit100ATM a1 = new Deposit100ATM(acc);
    Deposit100ATM a2 = new Deposit100ATM(acc);
    new Thread(a1).start();
    new Thread(a2).start();
    System.out.println("Amount on account is " + acc.getAmount());
}
```

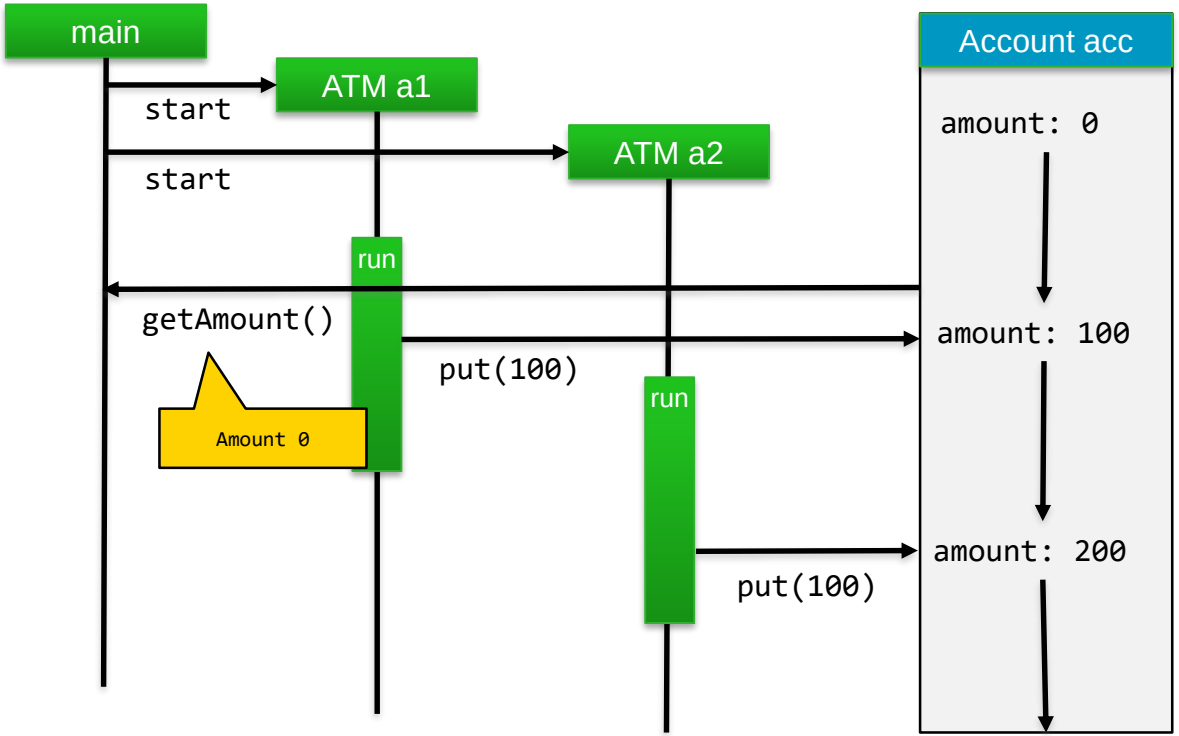
A POSSIBLE EXECUTION



A POSSIBLE EXECUTION



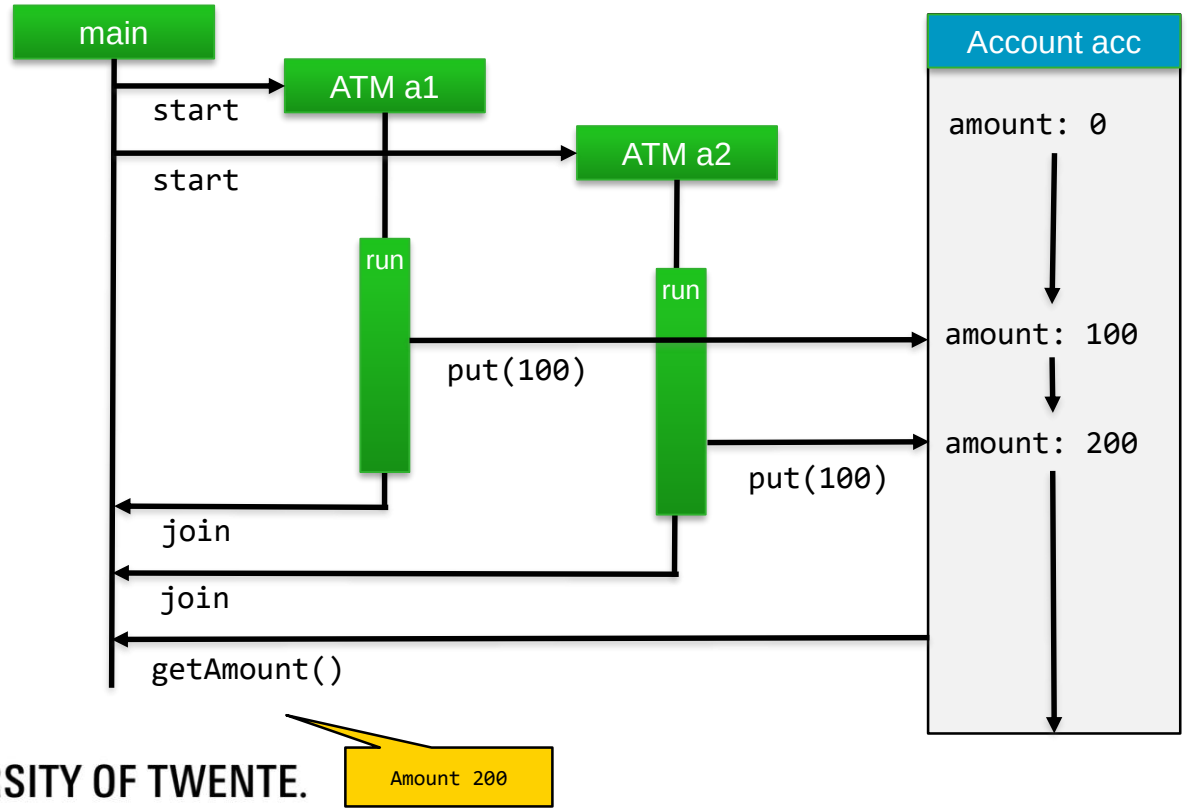
A POSSIBLE EXECUTION



WHAT WILL THIS PROGRAM PRINT?

```
public static void main(String[] args) {
    BankAccount acc = new BankAccount();
    Deposit100ATM a1 = new Deposit100ATM(acc);
    Deposit100ATM a2 = new Deposit100ATM(acc);
    a1.start();
    a2.start();
    try {
        a1.join();
        a2.join();
    } catch (InterruptedException e) { ... }
    System.out.println(”Amount on account is " + acc.getAmount());
}
```

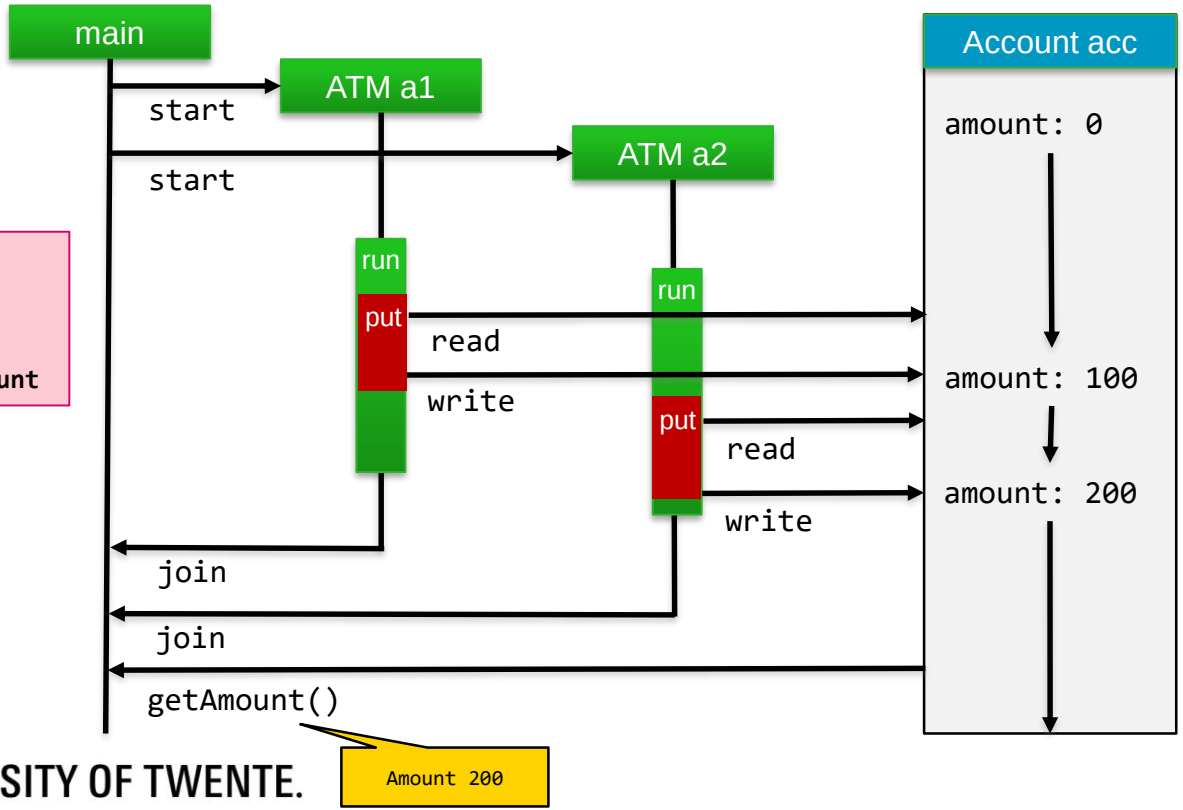
A POSSIBLE EXECUTION



A POSSIBLE EXECUTION

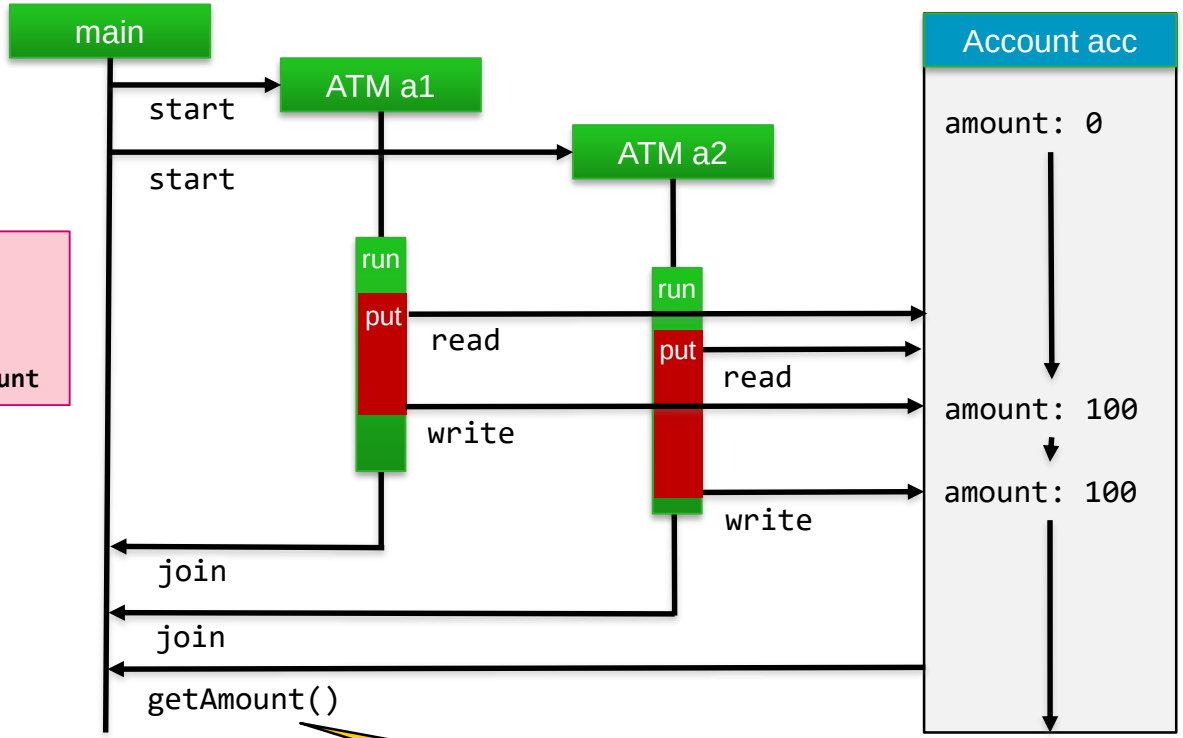
`amount = amount + val;`

1. Read `acc.amount`
2. Add `val` to `amount`
3. Assign to `acc.amount`



A POSSIBLE EXECUTION

```
amount = amount + val;
```
1. Read `acc.amount`
2. Add `val` to `amount`
3. Assign to `acc.amount`



This is called a data race
Should be avoided!

MUTUAL EXCLUSION FOR CRITICAL SECTION

- Critical section = code that ***must not*** be executed concurrently
  - Access to fields of shared objects
  - Use of external resources

- Example: the methods `deposit` and `withdraw` of BankAccount

- Solution: locks (also called mutexes, for mutual exclusion)
  - At most one thread at the time can hold the lock
  - Acquire the lock before the critical section
  - Release the lock after the critical section
  - Usually a thread is blocked until the lock becomes available

MUTUAL EXCLUSION FOR CRITICAL SECTION

- 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

MUTUAL EXCLUSION FOR CRITICAL SECTION

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`

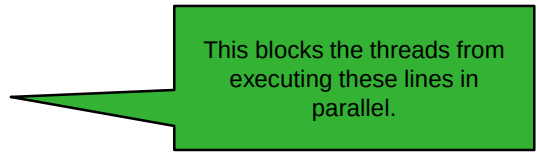
- Lock of `someObject` is passed to an arbitrary thread waiting for it

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!

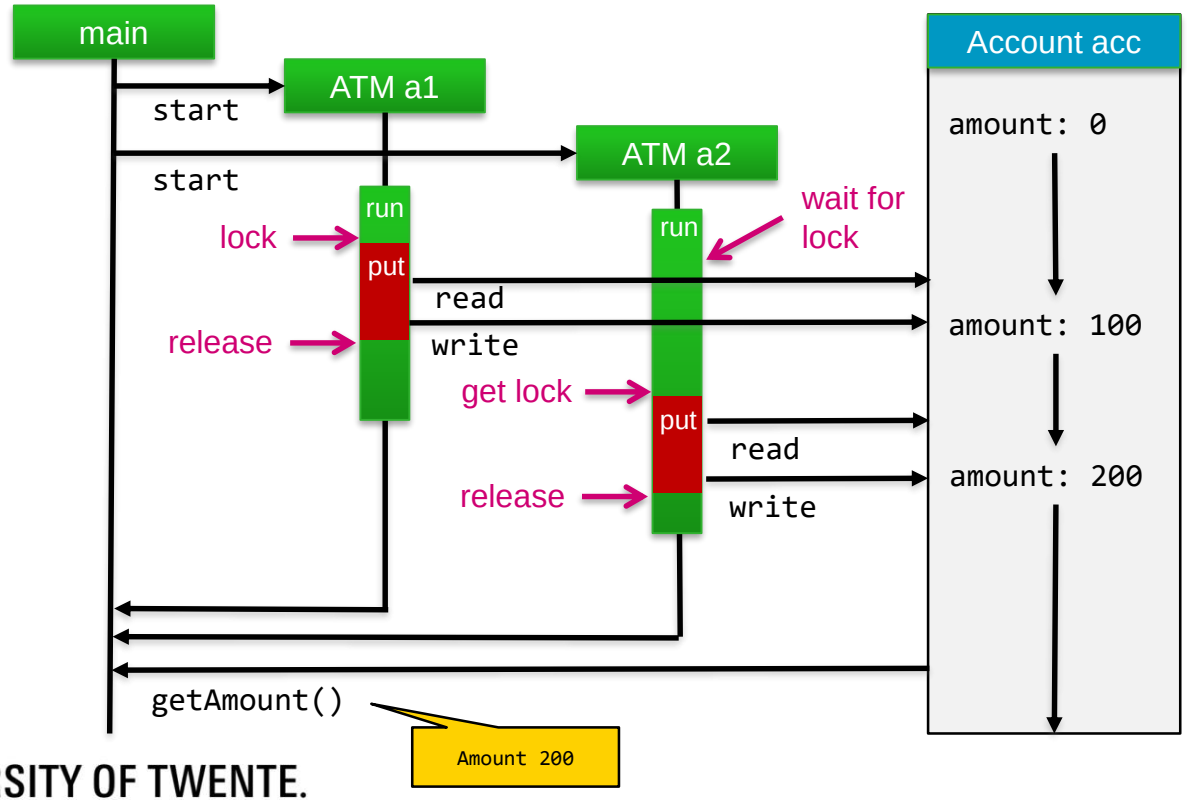
SYNCHRONIZE THE CALL

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



Client handles the synchronisation problem

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);
        }
    }
}
```

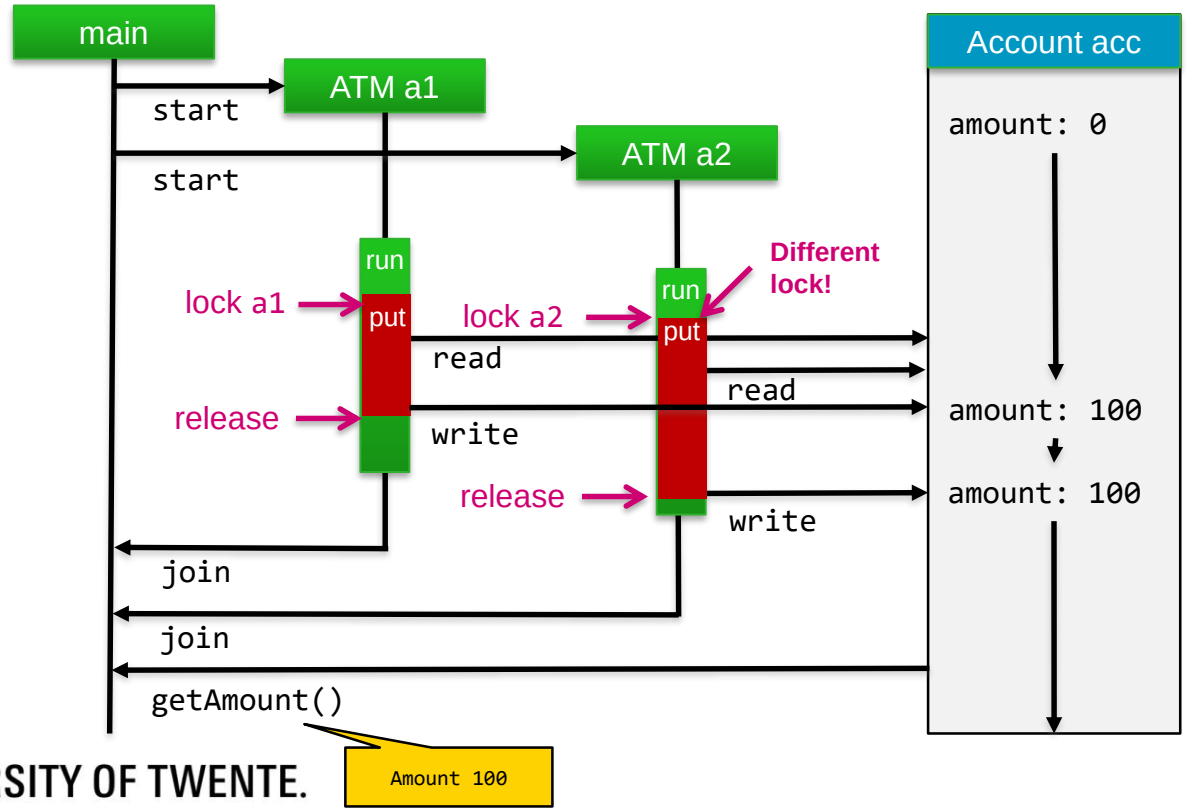
This will NOT work!

a1 and a2 will use different locks!

They have to use the same lock

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

A POSSIBLE EXECUTION



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`:

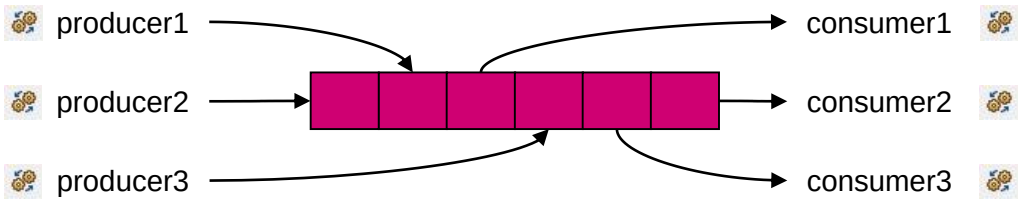
```
public class BankAccount {
    Lock l = new ReentrantLock();

    ...

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

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

WAITING FOR ANOTHER THREAD

Sometimes threads want to wait for a certain condition

Example: Producer and consumer threads with shared buffer



WAITING FOR A NON-EMPTY BUFFER

```
public class Consumer {
    ...
    // tries to read next element from buffer
    public Object getValue() {
        Object val = null;
        while (val == null) {
            synchronized (buffer) {
                if (!buffer.isEmpty()) {
                    val = buffer.getBuffer();
                }
            }
        }
        return val;
    }
}
```

Expensive ‘busy wait’ loop

COMMUNICATION BETWEEN THREADS: WAIT-NOTIFY

- Problem: How to let threads wait for certain conditions?

- Solution: conditions
  - A condition is associated with a mutex
    - Java: `lock.newCondition()` to create an implementation of the Condition interface
  - The wait or await operation waits until the condition is signalled
    - Release the mutex, then suspend the thread
    - After the thread is woken up, reacquire the mutex
  - The signal operation is called to indicate that the condition is now true
    - Typically (for example in Java) this wakes up 1 thread
    - Signal-and-continue: the signaling thread continues after signal (behavior of Java)
    - Signal-and-wait: the signaling thread yields to the signaled thread
  - The broadcast or signalAll operation wakes up all threads waiting for the condition

COMMUNICATION BETWEEN THREADS: WAIT-NOTIFY

- Java offers an easier method than using conditions directly: monitors
- A monitor combines the functionality of a mutex and a condition

- **Every** object in Java is also a monitor
  - `obj.wait()` releases mutex `obj` and wait
  - `obj.notify()` wakes up one arbitrary thread waiting for `obj`
  - `obj.notifyAll()` wakes up all threads waiting for `obj`

- These methods must only be called if a thread holds the lock

- `notifyAll`: more expensive
- `notify`: risk to wake up the ‘wrong’ thread

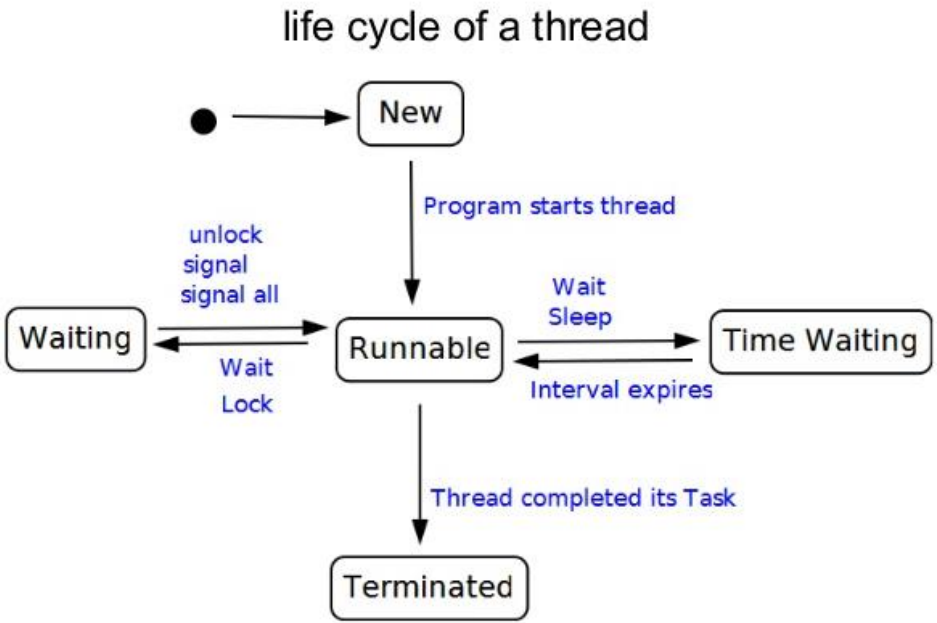
GETVALUE WITH WAIT-NOTIFY

```
public Object getValue() {
  synchronized (buffer) {
    while (buffer.isEmpty())
      try {
        buffer.wait();
      } catch (InterruptedException e) {
        ...
    }
    buffer.notifyAll();
    return buffer.getBuffer();
  }
}
```

Wait for a producer to add an item to the buffer

Wake all threads waiting for an event in the buffer

THREAD LIFECYCLE



TAKE HOME MESSAGES

- Multithreading: multiple threads execute in parallel
- Thread creation
  - Implement the Runnable interface
  - Create a thread and `start()` it and later wait for it to terminate with `join()`
- Threads share data (objects)
- Access to data should often be synchronized to avoid data races
  - Every object is a monitor (lock + condition)
  - Synchronized code blocks to lock/unlock around the block
  - Synchronized methods to lock/unlock around the method
  - More advanced: the Lock interface
- Inter-thread communication about object state
  - Methods `wait`, `notify`, `notifyAll`
  - More fine-grained: use conditions

More about concurrency: Programming Paradigms (Module 8)