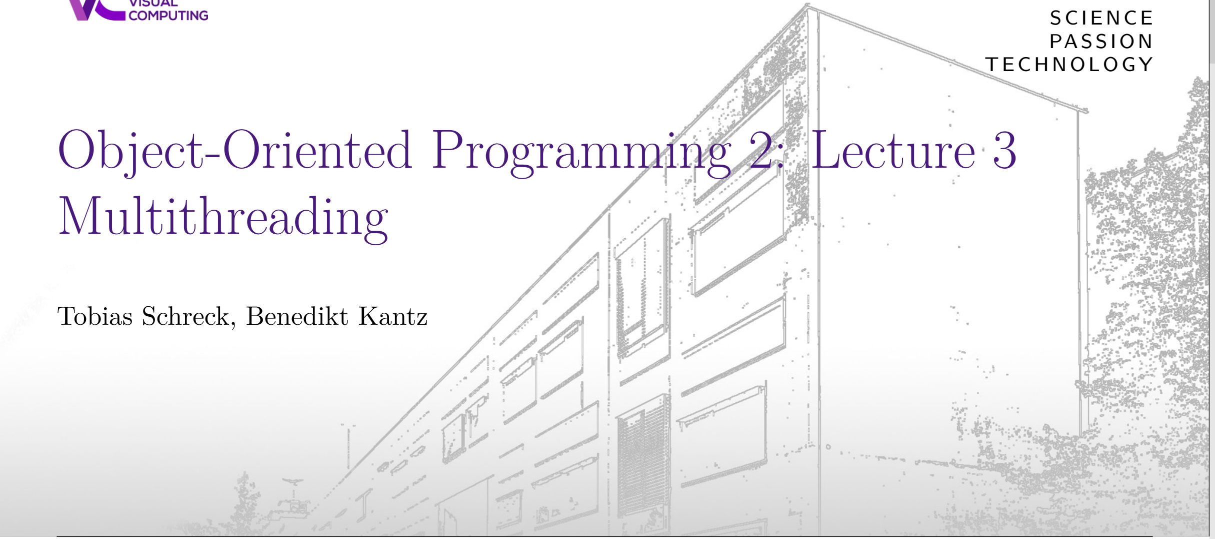
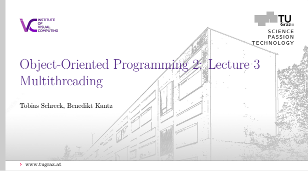


Object-Oriented Programming 2: Lecture 3 Multithreading

Tobias Schreck, Benedikt Kantz

2025-12-14

Object-Oriented Programming 2: Lecture 3 Multithreading



SCIENCE
PASSION
TECHNOLOGY

Object-Oriented Programming 2: Lecture 3
Multithreading

Tobias Schreck, Benedikt Kantz

www.tugraz.at

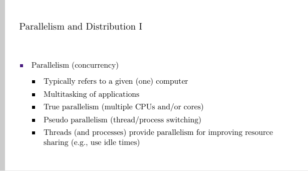
Which Multithreading and Synchronization frameworks/concepts do you know?

Which Multithreading and Synchronization frameworks/concepts do you know?

- Ask students, write down relevant answers on blackboard

Parallelism and Distribution I

- Parallelism (concurrency)
  - Typically refers to a given (one) computer
  - Multitasking of applications
  - True parallelism (multiple CPUs and/or cores)
  - Pseudo parallelism (thread/process switching)
  - Threads (and processes) provide parallelism for improving resource sharing (e.g., use idle times)



Parallelism and Distribution I

- Parallelism (concurrency)
  - Typically refers to a given (one) computer
  - Multitasking of applications
  - True parallelism (multiple CPUs and/or cores)
  - Pseudo parallelism (thread/process switching)
  - Threads (and processes) provide parallelism for improving resource sharing (e.g., use idle times)

Multitasking examples: on a desktop [download, streaming media, writing and email, update of weather widget,] within the desktop and among the Internet

Parallelism and Distribution II

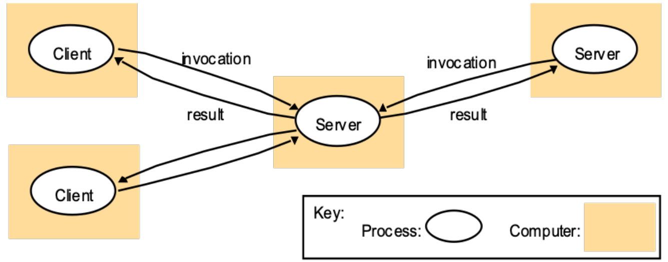
- Distributed system
  - Networked components, coordinated by messages
  - → Server processes handle different types of access (e.g., web, mail, file, ...)
  - → Server threads within server process handle concurrent user requests

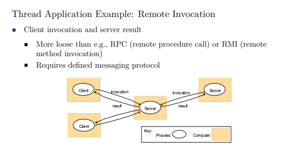
Parallelism and Distribution II

- Distributed system
  - Networked components, coordinated by messages
  - → Server processes handle different types of access (e.g., web, mail, file, ...)
  - → Server threads within server process handle concurrent user requests

Thread Application Example: Remote Invocation

- Client invocation and server result
  - More loose than e.g., RPC (remote procedure call) or RMI (remote method invocation)
  - Requires defined messaging protocol





Servers may be clients to other servers (e.g., web server using file server; search engine and crawler worker) Take advantage of consumer hardware capabilities (better than earlier servers)

More complex than client server (need to move and replicate objects and retrieve them)

Clients and Server with Threads

- Typical client side: User interface, processing logic, network communication
- Typical server side: Multiple clients, multiple communications

Clients and Server with Threads

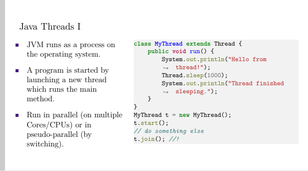
- Typical client side: User interface, processing logic, network communication
- Typical server side: Multiple clients, multiple communications

Es gibt am client und server typisch immer mehrere threads Threads leichter als Processe (kein Kontext Switch) Time slicing (interrups, scheduler) oder parallele ausfuehrung (multicore, hyperthreading - instruction-level parallelism per clock cycle)

Java Threads I

- JVM runs as a process on the operating system.
- A program is started by launching a new thread which runs the main method.
- Run in parallel (on multiple Cores/CPUs) or in pseudo-parallel (by switching).

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Hello from
        ↪  thread!");
        Thread.sleep(1000);
        System.out.println("Thread finished
        ↪  sleeping.");
    }
}
MyThread t = new MyThread();
t.start();
// do something else
t.join(); //!
```



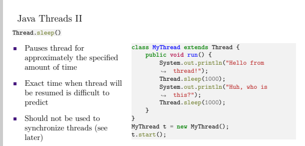
Threads can be implemented by objects which derive from java.lang.Thread
calling run directly would run the code not in a own thread but within the calling thread (a simple code branch)
Throws expection (interrupted)sequence and duration of thread execution varies, and may vary for different runs

Java Threads II

`Thread.sleep()`

- Pauses thread for approximately the specified amount of time
- Exact time when thread will be resumed is difficult to predict
- Should not be used to synchronize threads (see later)

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Hello from
        ↪  thread!");
        Thread.sleep(1000);
        System.out.println("Huh, who is
        ↪  this?");
        Thread.sleep(1000);
    }
}
MyThread t = new MyThread();
t.start();
```


Java Threads II

Thread.sleep()

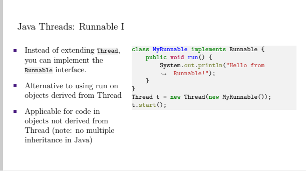
- Pauses thread for approximately the specified amount of time
- Exact time when thread will be resumed is difficult to predict
- Should not be used to synchronize threads (see later)

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Hello from
        ↪  thread!");
        Thread.sleep(1000);
        System.out.println("Huh, who is
        ↪  this?");
        Thread.sleep(1000);
    }
}
MyThread t = new MyThread();
t.start();
```

Java Threads: Runnable I

- Instead of extending `Thread`, you can implement the `Runnable` interface.
- Alternative to using run on objects derived from Thread
- Applicable for code in objects not derived from Thread (note: no multiple inheritance in Java)

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from
        ↪  Runnable!");
    }
}
Thread t = new Thread(new MyRunnable());
t.start();
```



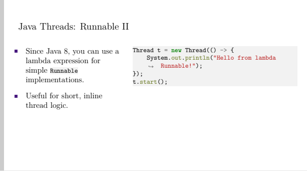
Java Threads: Runnable I

Implementing Runnable is preferred when you want to inherit from another class. The run() method contains the code executed by the thread. Lambda expressions Form: Parameterlist -> Code runnable is a functional interface Lambda expressions can be used where objects with a functional interface (ie an interface with only one abstract method) are required. (Note: since Java 8 there exist also non-abstract interface methods, called default methods).

Java Threads: Runnable II

- Since Java 8, you can use a lambda expression for simple `Runnable` implementations.
- Useful for short, inline thread logic.

```
Thread t = new Thread(() -> {
    System.out.println("Hello from lambda
    ↪  Runnable!");
});
t.start();
```

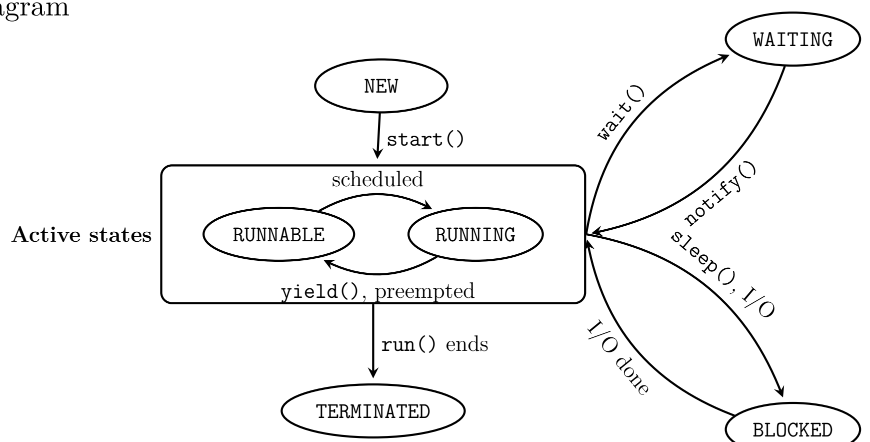

Java Threads: Runnable II

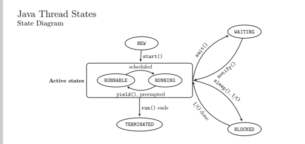
- Since Java 8, you can use a lambda expression for simple `Runnable` implementations.
- Useful for short, inline thread logic.

```
Thread t = new Thread(() -> {
    System.out.println("Hello from lambda
    ↪  Runnable!");
});
t.start();
```

Java Thread States

State Diagram





- **New**: Thread created, not yet started.
- **Runnable**: Ready to run, waiting for CPU.
- **Running**: Actively executing.
- **Blocked/Waiting**: Waiting for monitor, I/O, or sleep.
- **Terminated**: Finished execution.

Any problems with manually creating/assigning threads?

Any problems with manually creating/assigning threads?

- Ask students, write down relevant answers on blackboard

Thread Management

- solution: Executors

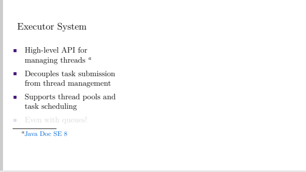
Thread Management

- solution: Executors

Executor System

- High-level API for managing threads [a]
- Decouples task submission from thread management
- Supports thread pools and task scheduling
- Even with queues!

[a] Java Doc SE 8


Executor System

- High-level API for managing threads [a]
- Decouples task submission from thread management
- Supports thread pools and task scheduling
- Even with queues!

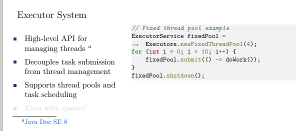
[a] Java Doc SE 8

Executor System

- High-level API for managing threads [a]
- Decouples task submission from thread management
- Supports thread pools and task scheduling
- Even with queues!

```java
// Fixed thread pool example
ExecutorService fixedPool =
↪   Executors.newFixedThreadPool(4);
for (int i = 0; i < 10; i++) {
    fixedPool.submit(() -> doWork());
}
fixedPool.shutdown();
```

[a] Java Doc SE 8



Executor System

- High-level API for managing threads [a]
- Decouples task submission from thread management
- Supports thread pools and task scheduling
- Even with queues!

```java
// Fixed thread pool example
ExecutorService fixedPool =
↪   Executors.newFixedThreadPool(4);
for (int i = 0; i < 10; i++) {
    fixedPool.submit(() -> doWork());
}
fixedPool.shutdown();
```

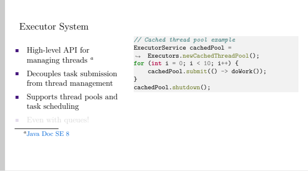
[a] Java Doc SE 8

Executor System

- High-level API for managing threads [a]
- Decouples task submission from thread management
- Supports thread pools and task scheduling
- Even with queues!

```java
// Cached thread pool example
ExecutorService cachedPool =
↪  Executors.newCachedThreadPool();
for (int i = 0; i < 10; i++) {
    cachedPool.submit(() -> doWork());
}
cachedPool.shutdown();
```

[a] Java Doc SE 8


Executor System

- High-level API for managing threads [a]
- Decouples task submission from thread management
- Supports thread pools and task scheduling
- Even with queues!

```java
// Cached thread pool example
ExecutorService cachedPool =
↪  Executors.newCachedThreadPool();
for (int i = 0; i < 10; i++) {
    cachedPool.submit(() -> doWork());
}
cachedPool.shutdown();
```

[a] Java Doc SE 8

Executor System

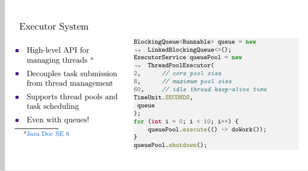
- High-level API for managing threads [a]
- Decouples task submission from thread management
- Supports thread pools and task scheduling
- Even with queues!

```
BlockingQueue<Runnable> queue = new
↪  LinkedBlockingQueue<>();
ExecutorService queuePool = new
↪  ThreadPoolExecutor(
2,      // core pool size
8,      // maximum pool size
60,     // idle thread keep-alive time
TimeUnit.SECONDS,
 queue
);
for (int i = 0; i < 10; i++) {
    queuePool.execute(() -> doWork());
}
queuePool.shutdown();
```

[a] Java Doc SE 8

2025-12-14

Object-Oriented Programming 2: Lecture 3 Multithreading
└Managament
  └Executor System


Executor System

- High-level API for managing threads [a]
- Decouples task submission from thread management
- Supports thread pools and task scheduling
- Even with queues!

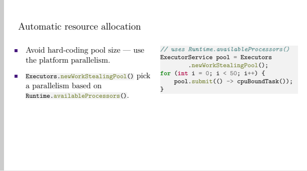
```
BlockingQueue<Runnable> queue = new
↪  LinkedBlockingQueue<>();
ExecutorService queuePool = new
↪  ThreadPoolExecutor(
2,      // core pool size
8,      // maximum pool size
60,     // idle thread keep-alive time
TimeUnit.SECONDS,
 queue
);
for (int i = 0; i < 10; i++) {
    queuePool.execute(() -> doWork());
}
queuePool.shutdown();
```

[a] Java Doc SE 8

Automatic resource allocation

- Avoid hard-coding pool size — use the platform parallelism.
- `Executors.newWorkStealingPool()` pick a parallelism based on `Runtime.availableProcessors()`.

```
// uses Runtime.availableProcessors()
ExecutorService pool = Executors
        .newWorkStealingPool();
for (int i = 0; i < 50; i++) {
    pool.submit(() -> cpuBoundTask());
}
```


Automatic resource allocation

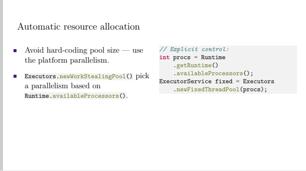
- Avoid hard-coding pool size — use the platform parallelism.
- `Executors.newWorkStealingPool()` pick a parallelism based on `Runtime.availableProcessors()`.

```
// uses Runtime.availableProcessors()
ExecutorService pool = Executors
        .newWorkStealingPool();
for (int i = 0; i < 50; i++) {
    pool.submit(() -> cpuBoundTask());
}
```

Automatic resource allocation

- Avoid hard-coding pool size — use the platform parallelism.
- `Executors.newWorkStealingPool()` pick a parallelism based on `Runtime.availableProcessors()`.

```java
// Explicit control:
int procs = Runtime
    .getRuntime()
    .availableProcessors();
ExecutorService fixed = Executors
    .newFixedThreadPool(procs);
```



Automatic resource allocation

- Avoid hard-coding pool size — use the platform parallelism.
- `Executors.newWorkStealingPool()` pick a parallelism based on `Runtime.availableProcessors()`.

```java
// Explicit control:
int procs = Runtime
    .getRuntime()
    .availableProcessors();
ExecutorService fixed = Executors
    .newFixedThreadPool(procs);
```

Automatic resource allocation

- Avoid hard-coding pool size — use the platform parallelism.
- `Executors.newWorkStealingPool()` pick a parallelism based on `Runtime.availableProcessors()`.

```java
// Explicit control:
int procs = Runtime
    .getRuntime()
    .availableProcessors();
ExecutorService fixed = Executors
    .newFixedThreadPool(procs);
```

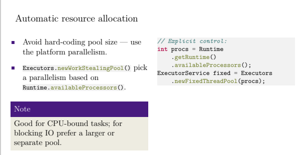
Note

Good for CPU-bound tasks; for blocking IO prefer a larger or separate pool.

2025-12-14

Object-Oriented Programming 2: Lecture 3 Multithreading
└Managament

└Automatic resource allocation


Automatic resource allocation

- Avoid hard-coding pool size — use the platform parallelism.
- `Executors.newWorkStealingPool()` pick a parallelism based on `Runtime.availableProcessors()`.

```java
// Explicit control:
int procs = Runtime
    .getRuntime()
    .availableProcessors();
ExecutorService fixed = Executors
    .newFixedThreadPool(procs);
```

Note

Good for CPU-bound tasks; for blocking IO prefer a larger or separate pool.

Which Multithreading and Synchronization frameworks concepts have you used?

Which Multithreading and Synchronization frameworks concepts have you used?

- Ask students, write down relevant answers on blackboard

Synchronization

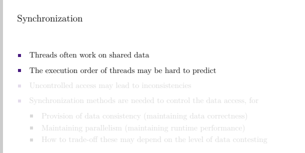
- Threads often work on shared data
- The execution order of threads may be hard to predict
- Uncontrolled access may lead to inconsistencies
- Synchronization methods are needed to control the data access, for
  - Provision of data consistency (maintaining data correctness)
  - Maintaining parallelism (maintaining runtime performance)
  - How to trade-off these may depend on the level of data contesting

Synchronization

- Threads often work on shared data
- The execution order of threads may be hard to predict
- Uncontrolled access may lead to inconsistencies
- Synchronization methods are needed to control the data access, for
  - Provision of data consistency (maintaining data correctness)
  - Maintaining parallelism (maintaining runtime performance)
  - How to trade-off these may depend on the level of data contesting

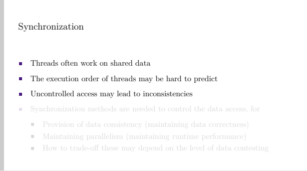
Synchronization

- Threads often work on shared data
- The execution order of threads may be hard to predict
- Uncontrolled access may lead to inconsistencies
- Synchronization methods are needed to control the data access, for
  - Provision of data consistency (maintaining data correctness)
  - Maintaining parallelism (maintaining runtime performance)
  - How to trade-off these may depend on the level of data contesting


Synchronization
- Threads often work on shared data
- The execution order of threads may be hard to predict
- Uncontrolled access may lead to inconsistencies
- Synchronization methods are needed to control the data access, for
  - Provision of data consistency (maintaining data correctness)
  - Maintaining parallelism (maintaining runtime performance)
  - How to trade-off these may depend on the level of data contesting

Synchronization

- Threads often work on shared data
- The execution order of threads may be hard to predict
- Uncontrolled access may lead to inconsistencies
- Synchronization methods are needed to control the data access, for
  - Provision of data consistency (maintaining data correctness)
  - Maintaining parallelism (maintaining runtime performance)
  - How to trade-off these may depend on the level of data contesting



Synchronization

- Threads often work on shared data
- The execution order of threads may be hard to predict
- Uncontrolled access may lead to inconsistencies
- Synchronization methods are needed to control the data access, for
  - Provision of data consistency (maintaining data correctness)
  - Maintaining parallelism (maintaining runtime performance)
  - How to trade-off these may depend on the level of data contesting

Synchronization

- Threads often work on shared data
- The execution order of threads may be hard to predict
- Uncontrolled access may lead to inconsistencies
- Synchronization methods are needed to control the data access, for
  - Provision of data consistency (maintaining data correctness)
  - Maintaining parallelism (maintaining runtime performance)
  - How to trade-off these may depend on the level of data contesting

Synchronization

- Threads often work on shared data
- The execution order of threads may be hard to predict
- Uncontrolled access may lead to inconsistencies
- Synchronization methods are needed to control the data access, for
  - Provision of data consistency (maintaining data correctness)
  - Maintaining parallelism (maintaining runtime performance)
  - How to trade-off these may depend on the level of data contesting

Synchronization
Example

- Conflicting access to a bank account (see code example).
- Problem: Without specific controls, we cannot predict where a thread switch (or true parallel access) may occur.

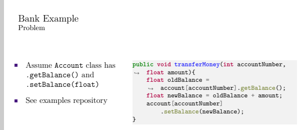
Synchronization
Example

- Conflicting access to a bank account (see code example).
- Problem: Without specific controls, we cannot predict where a thread switch (or true parallel access) may occur.

Bank Example
Problem

- Assume `Account` class has `.getBalance()` and `.setBalance(float)`
- See examples repository

```
public void transferMoney(int accountNumber,
↪   float amount){
    float oldBalance =
    ↪   account[accountNumber].getBalance();
    float newBalance = oldBalance + amount;
    account[accountNumber]
        .setBalance(newBalance);
}
```


Bank Example
Problem

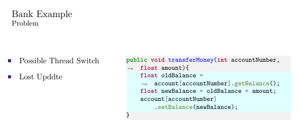
- Assume `Account` class has `.getBalance()` and `.setBalance(float)`
- See examples repository

```
public void transferMoney(int accountNumber,
↪   float amount){
    float oldBalance =
    ↪   account[accountNumber].getBalance();
    float newBalance = oldBalance + amount;
    account[accountNumber]
        .setBalance(newBalance);
}
```

Bank Example
Problem

- Possible Thread Switch
- Lost Upddte

```
public void transferMoney(int accountNumber,
↪  float amount){
    float oldBalance =
    ↪  account[accountNumber].getBalance();
    float newBalance = oldBalance + amount;
    account[accountNumber]
        .setBalance(newBalance);
}
```


Bank Example
Problem

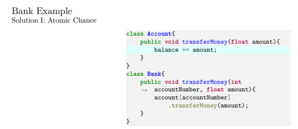
- Possible Thread Switch
- Lost Upddte

```
public void transferMoney(int accountNumber,
↪  float amount){
    float oldBalance =
    ↪  account[accountNumber].getBalance();
    float newBalance = oldBalance + amount;
    account[accountNumber]
        .setBalance(newBalance);
}
```

Bank Example

Solution I: Atomic Chance

```
class Account{
    public void transferMoney(float amount){
        balance += amount;
    }
}
class Bank{
    public void transferMoney(int
    ↪  accountNumber, float amount){
        account[accountNumber]
            .transferMoney(amount);
    }
}
```

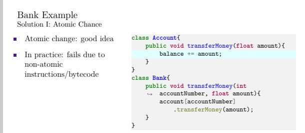


There are instructions at system level for x86 that *are* atomic - the example would nevertheless fail as the transaction itself is not atomic

Bank Example
Solution I: Atomic Chance

- Atomic change: good idea
- In practice: fails due to non-atomic instructions/bytecode

```
class Account{
    public void transferMoney(float amount){
        balance += amount;
    }
}
class Bank{
    public void transferMoney(int
    ↪ accountNumber, float amount){
        account[accountNumber]
            .transferMoney(amount);
    }
}
```

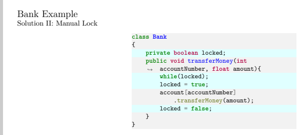


There are instructions at system level for x86 that *are* atomic - the example would nevertheless fail as the transaction itself is not atomic

Bank Example

Solution II: Manual Lock

```
class Bank
{
    private boolean locked;
    public void transferMoney(int
     ↪  accountNumber, float amount){
        while(locked);
        locked = true;
        account[accountNumber]
            .transferMoney(amount);
        locked = false;
    }
}
```


Bank Example
Solution II: Manual Lock

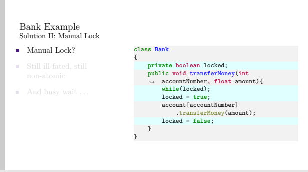
```
class Bank
{
    private boolean locked;
    public void transferMoney(int
     ↪  accountNumber, float amount){
        while(locked);
        locked = true;
        account[accountNumber]
            .transferMoney(amount);
        locked = false;
    }
}
```

Bank Example

Solution II: Manual Lock

- Manual Lock?
- Still ill-fated, still non-atomic
- And busy wait …

```
class Bank
{
    private boolean locked;
    public void transferMoney(int
    ↪   accountNumber, float amount){
        while(locked);
        locked = true;
        account[accountNumber]
            .transferMoney(amount);
        locked = false;
    }
}
```


Bank Example
Solution II: Manual Lock

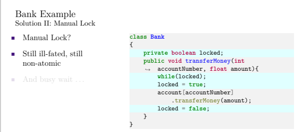
- Manual Lock?
- Still ill-fated, still non-atomic
- And busy wait …

```
class Bank
{
    private boolean locked;
    public void transferMoney(int
    ↪   accountNumber, float amount){
        while(locked);
        locked = true;
        account[accountNumber]
            .transferMoney(amount);
        locked = false;
    }
}
```

Bank Example
Solution II: Manual Lock

- Manual Lock?
- Still ill-fated, still non-atomic
- And busy wait ...

```
class Bank
{
    private boolean locked;
    public void transferMoney(int
    ↪   accountNumber, float amount){
        while(locked);
        locked = true;
        account[accountNumber]
            .transferMoney(amount);
        locked = false;
    }
}
```


Bank Example
Solution II: Manual Lock

- Manual Lock?
- Still ill-fated, still non-atomic
- And busy wait ...

```
class Bank
{
    private boolean locked;
    public void transferMoney(int
    ↪   accountNumber, float amount){
        while(locked);
        locked = true;
        account[accountNumber]
            .transferMoney(amount);
        locked = false;
    }
}
```

Bank Example

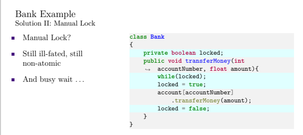
Solution II: Manual Lock

- Manual Lock?
- Still ill-fated, still non-atomic
- And busy wait ...

```
class Bank
{
    private boolean locked;
    public void transferMoney(int
    ↪   accountNumber, float amount){
        while(locked);
        locked = true;
        account[accountNumber]
            .transferMoney(amount);
        locked = false;
    }
}
```

2025-12-14

Object-Oriented Programming 2: Lecture 3 Multithreading
└Synchronization
└Bank Example

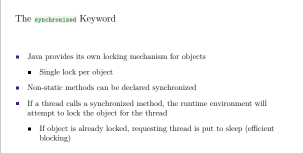

Bank Example
Solution II: Manual Lock

- Manual Lock?
- Still ill-fated, still non-atomic
- And busy wait ...

```
class Bank
{
    private boolean locked;
    public void transferMoney(int
    ↪   accountNumber, float amount){
        while(locked);
        locked = true;
        account[accountNumber]
            .transferMoney(amount);
        locked = false;
    }
}
```

The `synchronized` Keyword

- Java provides its own locking mechanism for objects
  - Single lock per object
- Non-static methods can be declared synchronized
- If a thread calls a synchronized method, the runtime environment will attempt to lock the object for the thread
  - If object is already locked, requesting thread is put to sleep (efficient blocking)


The `synchronized` Keyword

- Java provides its own locking mechanism for objects
  - Single lock per object
- Non-static methods can be declared synchronized
- If a thread calls a synchronized method, the runtime environment will attempt to lock the object for the thread
  - If object is already locked, requesting thread is put to sleep (efficient blocking)

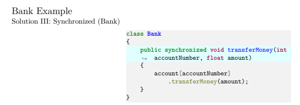
- Should be used at appropriate grain (don’t lock just everything)
- Java runtime overhead for lock management
- Deadlocks may occur

Frage: warum nur non-static methods? (nur diese haben ein eigenes Java Lock Feld)

Bank Example

Solution III: Synchronized (Bank)

```
class Bank
{
    public synchronized void transferMoney(int
    ↪  accountNumber, float amount)
    {
        account[accountNumber]
            .transferMoney(amount);
    }
}
```


Bank Example
Solution III: Synchronized (Bank)

```
class Bank
{
    public synchronized void transferMoney(int
    ↪  accountNumber, float amount)
    {
        account[accountNumber]
            .transferMoney(amount);
    }
}
```

Bank Example

Solution III: Synchronized (Bank)

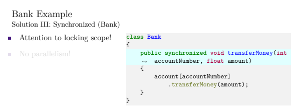
- Attention to locking scope!
- No parallelism!

```
class Bank
{
    public synchronized void transferMoney(int
    ↪  accountNumber, float amount)
    {
        account[accountNumber]
            .transferMoney(amount);
    }
}
```

2025-12-14

Object-Oriented Programming 2: Lecture 3 Multithreading
└Synchronization

└Bank Example



Bank Example
Solution III: Synchronized (Bank)

- Attention to locking scope!
- No parallelism!

```
class Bank
{
    public synchronized void transferMoney(int
    ↪  accountNumber, float amount)
    {
        account[accountNumber]
            .transferMoney(amount);
    }
}
```

Bank Example

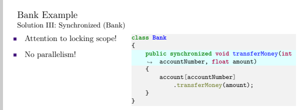
Solution III: Synchronized (Bank)

- Attention to locking scope!
- No parallelism!

```
class Bank
{
    public synchronized void transferMoney(int
    ↪  accountNumber, float amount)
    {
        account[accountNumber]
            .transferMoney(amount);
    }
}
```

2025-12-14

Object-Oriented Programming 2: Lecture 3 Multithreading
└Synchronization
└Bank Example


Bank Example
Solution III: Synchronized (Bank)

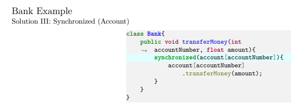
- Attention to locking scope!
- No parallelism!

```
class Bank
{
    public synchronized void transferMoney(int
    ↪  accountNumber, float amount)
    {
        account[accountNumber]
            .transferMoney(amount);
    }
}
```

Bank Example

Solution III: Synchronized (Account)

```
class Bank{
    public void transferMoney(int
    ↪   accountNumber, float amount){
        synchronized(account[accountNumber]){
            account[accountNumber]
                .transferMoney(amount);
        }
    }
}
```

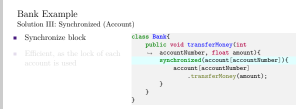


Bank Example

Solution III: Synchronized (Account)

- Synchronize block
- Efficient, as the lock of each account is used

```
class Bank{
    public void transferMoney(int
    ↪   accountNumber, float amount){
        synchronized(account[accountNumber]){
            account[accountNumber]
                .transferMoney(amount);
        }
    }
}
```

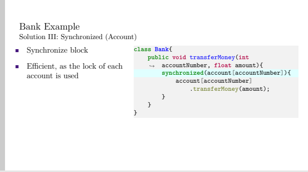


Bank Example

Solution III: Synchronized (Account)

- Synchronize block
- Efficient, as the lock of each account is used

```
class Bank{
    public void transferMoney(int
    ↪  accountNumber, float amount){
        synchronized(account[accountNumber]){
            account[accountNumber]
                .transferMoney(amount);
        }
    }
}
```


Bank Example
Solution III: Synchronized (Account)

- Synchronize block
- Efficient, as the lock of each account is used

```
class Bank{
    public void transferMoney(int
    ↪  accountNumber, float amount){
        synchronized(account[accountNumber]){
            account[accountNumber]
                .transferMoney(amount);
        }
    }
}
```

Do you know what a semaphore is?

Do you know what a semaphore is?

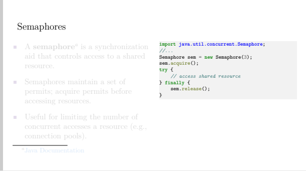
- Ask students, write down relevant answers on blackboard

Semaphores

- A **semaphore**[a] is a synchronization aid that controls access to a shared resource.
- Semaphores maintain a set of permits; acquire permits before accessing resources.
- Useful for limiting the number of concurrent accesses a resource (e.g., connection pools).

```
import java.util.concurrent.Semaphore;
//...
Semaphore sem = new Semaphore(3);
sem.acquire();
try {
    // access shared resource
} finally {
    sem.release();
}
```

[a] Java Documentation


Semaphores

- A **semaphore**[a] is a synchronization aid that controls access to a shared resource.
- Semaphores maintain a set of permits; acquire permits before accessing resources.
- Useful for limiting the number of concurrent accesses a resource (e.g., connection pools).

```
import java.util.concurrent.Semaphore;
//...
Semaphore sem = new Semaphore(3);
sem.acquire();
try {
    // access shared resource
} finally {
    sem.release();
}
```

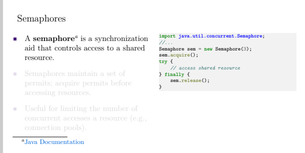
[a] Java Documentation

Semaphores

- A **semaphore**[a] is a synchronization aid that controls access to a shared resource.
- Semaphores maintain a set of permits; acquire permits before accessing resources.
- Useful for limiting the number of concurrent accesses a resource (e.g., connection pools).

```java
import java.util.concurrent.Semaphore;
//...
Semaphore sem = new Semaphore(3);
sem.acquire();
try {
    // access shared resource
} finally {
    sem.release();
}
```

[a] Java Documentation


Semaphores

- A **semaphore**[a] is a synchronization aid that controls access to a shared resource.
- Semaphores maintain a set of permits; acquire permits before accessing resources.
- Useful for limiting the number of concurrent accesses a resource (e.g., connection pools).

```java
import java.util.concurrent.Semaphore;
//...
Semaphore sem = new Semaphore(3);
sem.acquire();
try {
    // access shared resource
} finally {
    sem.release();
}
```

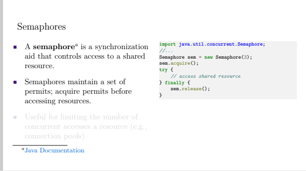
[a] Java Documentation

Semaphores

- A **semaphore**[a] is a synchronization aid that controls access to a shared resource.
- Semaphores maintain a set of permits; acquire permits before accessing resources.
- Useful for limiting the number of concurrent accesses a resource (e.g., connection pools).

```java
import java.util.concurrent.Semaphore;
//...
Semaphore sem = new Semaphore(3);
sem.acquire();
try {
    // access shared resource
} finally {
    sem.release();
}
```

[a] Java Documentation


Semaphores

- A **semaphore**[a] is a synchronization aid that controls access to a shared resource.
- Semaphores maintain a set of permits; acquire permits before accessing resources.
- Useful for limiting the number of concurrent accesses a resource (e.g., connection pools).

```java
import java.util.concurrent.Semaphore;
//...
Semaphore sem = new Semaphore(3);
sem.acquire();
try {
    // access shared resource
} finally {
    sem.release();
}
```

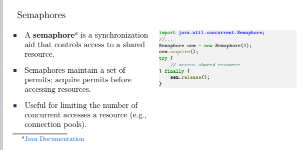
[a] Java Documentation

Semaphores

- A **semaphore**[a] is a synchronization aid that controls access to a shared resource.
- Semaphores maintain a set of permits; acquire permits before accessing resources.
- Useful for limiting the number of concurrent accesses a resource (e.g., connection pools).

```java
import java.util.concurrent.Semaphore;
//...
Semaphore sem = new Semaphore(3);
sem.acquire();
try {
    // access shared resource
} finally {
    sem.release();
}
```

[a] Java Documentation

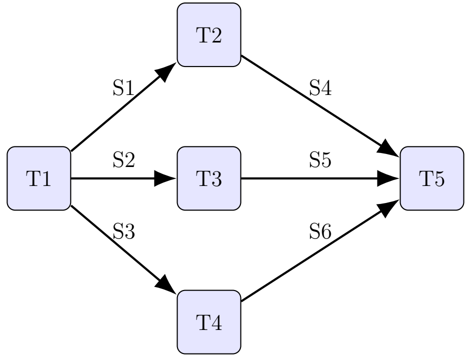

Semaphores

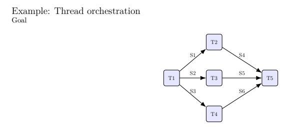
- A **semaphore**[a] is a synchronization aid that controls access to a shared resource.
- Semaphores maintain a set of permits; acquire permits before accessing resources.
- Useful for limiting the number of concurrent accesses a resource (e.g., connection pools).

```java
import java.util.concurrent.Semaphore;
//...
Semaphore sem = new Semaphore(3);
sem.acquire();
try {
    // access shared resource
} finally {
    sem.release();
}
```

[a] Java Documentation

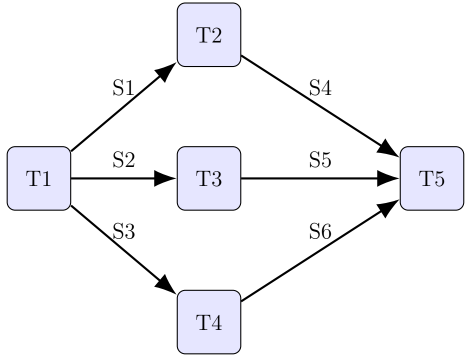
Example: Thread orchestration
Goal


$$\begin{tikzcd}
 & T2 \arrow[dr, "S4"] & \\
T1 \arrow[ur, "S1"] \arrow[r, "S2"] \arrow[dr, "S3"'] & T3 \arrow[r, "S5"] & T5 \\
 & T4 \arrow[ur, "S6"'] &
\end{tikzcd}$$

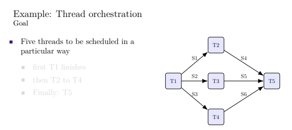


Example: Thread orchestration
Goal

- Five threads to be scheduled in a particular way
  - first T1 finishes
  - then T2 to T4
  - Finally: T5

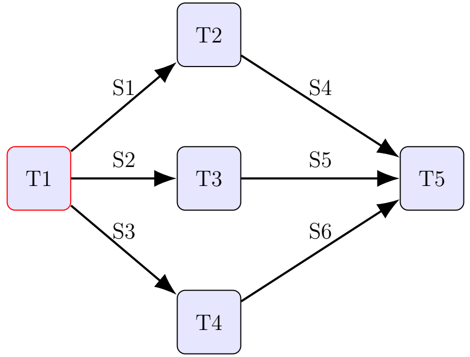


$$\begin{tikzcd}
 & T2 \arrow[dr, "S4"] & \\
T1 \arrow[ur, "S1"] \arrow[r, "S2"] \arrow[dr, "S3"'] & T3 \arrow[r, "S5"] & T5 \\
 & T4 \arrow[ur, "S6"'] &
\end{tikzcd}$$

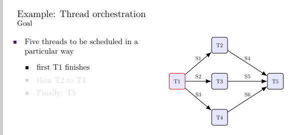


Example: Thread orchestration
Goal

- Five threads to be scheduled in a particular way
  - first T1 finishes
  - then T2 to T4
  - Finally: T5

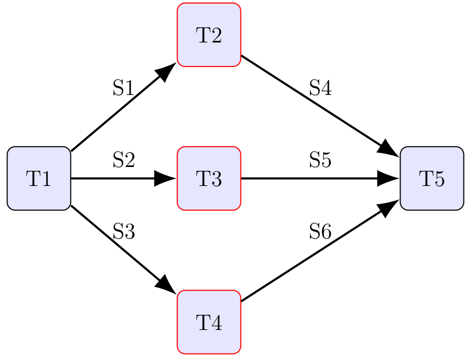


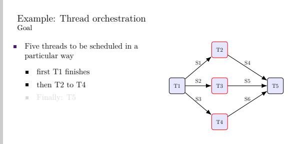
$$\begin{tikzcd}
 & T2 \arrow[dr, "S4"] & \\
T1 \arrow[ur, "S1"] \arrow[r, "S2"] \arrow[dr, "S3"'] & T3 \arrow[r, "S5"] & T5 \\
 & T4 \arrow[ur, "S6"'] &
\end{tikzcd}$$



Example: Thread orchestration
Goal

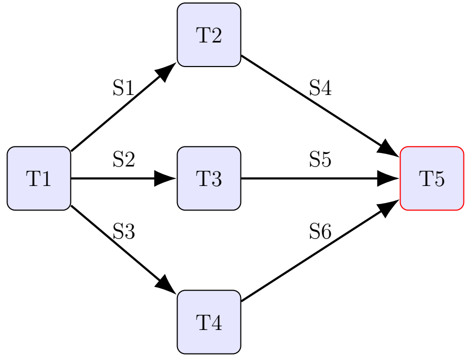
- Five threads to be scheduled in a particular way
  - first T1 finishes
  - then T2 to T4
  - Finally: T5

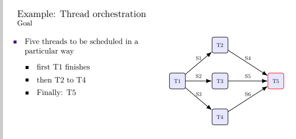

$$\begin{tikzcd}
 & T2 \arrow[dr, "S4"] & \\
T1 \arrow[ur, "S1"] \arrow[r, "S2"] \arrow[dr, "S3"'] & T3 \arrow[r, "S5"] & T5 \\
 & T4 \arrow[ur, "S6"'] &
\end{tikzcd}$$



Example: Thread orchestration
Goal

- Five threads to be scheduled in a particular way
  - first T1 finishes
  - then T2 to T4
  - Finally: T5



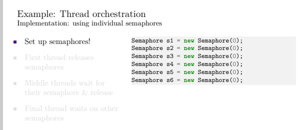


Example: Thread orchestration

Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

```
Semaphore s1 = new Semaphore(0);
Semaphore s2 = new Semaphore(0);
Semaphore s3 = new Semaphore(0);
Semaphore s4 = new Semaphore(0);
Semaphore s5 = new Semaphore(0);
Semaphore s6 = new Semaphore(0);
```


Example: Thread orchestration
Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

```
Semaphore s1 = new Semaphore(0);
Semaphore s2 = new Semaphore(0);
Semaphore s3 = new Semaphore(0);
Semaphore s4 = new Semaphore(0);
Semaphore s5 = new Semaphore(0);
Semaphore s6 = new Semaphore(0);
```

Example: Thread orchestration

Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

```java
Thread t1 = new Thread(() -> {
    System.out.println("T1 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T1 finished");
    s1.release();
    s2.release();
    s3.release();
});
```

Example: Thread orchestration

Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

```java
Thread t2 = new Thread(() -> {
    s1.acquire();
    System.out.println("T2 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T2 finished");
    s4.release();
});
```

Example: Thread orchestration
Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

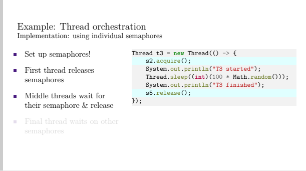
```java
Thread t2 = new Thread(() -> {
    s1.acquire();
    System.out.println("T2 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T2 finished");
    s4.release();
});
```

Example: Thread orchestration

Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

```java
Thread t3 = new Thread(() -> {
    s2.acquire();
    System.out.println("T3 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T3 finished");
    s5.release();
});
```


Example: Thread orchestration
Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

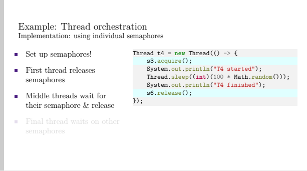
```java
Thread t3 = new Thread(() -> {
    s2.acquire();
    System.out.println("T3 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T3 finished");
    s5.release();
});
```

Example: Thread orchestration

Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

```
Thread t4 = new Thread(() -> {
    s3.acquire();
    System.out.println("T4 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T4 finished");
    s6.release();
});
```

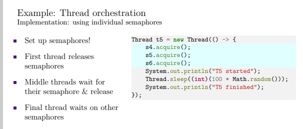


Example: Thread orchestration

Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

```java
Thread t5 = new Thread(() -> {
    s4.acquire();
    s5.acquire();
    s6.acquire();
    System.out.println("T5 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T5 finished");
});
```


Example: Thread orchestration
Implementation: using individual semaphores

- Set up semaphores!
- First thread releases semaphores
- Middle threads wait for their semaphore & release
- Final thread waits on other semaphores

```java
Thread t5 = new Thread(() -> {
    s4.acquire();
    s5.acquire();
    s6.acquire();
    System.out.println("T5 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T5 finished");
});
```

Example: Thread orchestration
Implementation

(How) Can we make this more efficient?

Example: Thread orchestration
Implementation

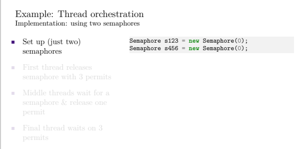
(How) Can we make this more efficient?

Example: Thread orchestration

Implementation: using two semaphores

- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```
Semaphore s123 = new Semaphore(0);
Semaphore s456 = new Semaphore(0);
```



Example: Thread orchestration
Implementation: using two semaphores

- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

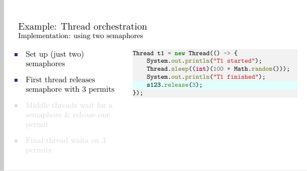
```
Semaphore s123 = new Semaphore(0);
Semaphore s456 = new Semaphore(0);
```

Example: Thread orchestration

Implementation: using two semaphores

- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```java
Thread t1 = new Thread(() -> {
    System.out.println("T1 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T1 finished");
    s123.release(3);
});
```


Example: Thread orchestration
Implementation: using two semaphores

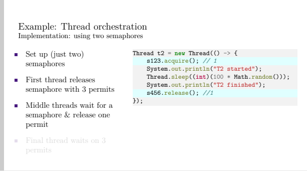
- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```java
Thread t1 = new Thread(() -> {
    System.out.println("T1 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T1 finished");
    s123.release(3);
});
```

Example: Thread orchestration
Implementation: using two semaphores

- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```java
Thread t2 = new Thread(() -> {
    s123.acquire(); // 1
    System.out.println("T2 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T2 finished");
    s456.release(); //1
});
```


Example: Thread orchestration
Implementation: using two semaphores

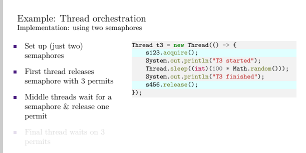
- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```java
Thread t2 = new Thread(() -> {
    s123.acquire(); // 1
    System.out.println("T2 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T2 finished");
    s456.release(); //1
});
```

Example: Thread orchestration
Implementation: using two semaphores

- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```
Thread t3 = new Thread(() -> {
    s123.acquire();
    System.out.println("T3 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T3 finished");
    s456.release();
});
```


Example: Thread orchestration
Implementation: using two semaphores
- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

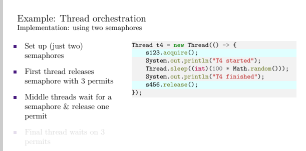
```
Thread t3 = new Thread(() -> {
    s123.acquire();
    System.out.println("T3 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T3 finished");
    s456.release();
});
```

Example: Thread orchestration

Implementation: using two semaphores

- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```java
Thread t4 = new Thread(() -> {
    s123.acquire();
    System.out.println("T4 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T4 finished");
    s456.release();
});
```


Example: Thread orchestration
Implementation: using two semaphores

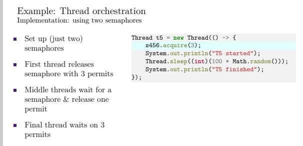
- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```java
Thread t4 = new Thread(() -> {
    s123.acquire();
    System.out.println("T4 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T4 finished");
    s456.release();
});
```

Example: Thread orchestration
Implementation: using two semaphores

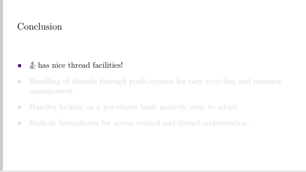
- Set up (just two) semaphores
- First thread releases semaphore with 3 permits
- Middle threads wait for a semaphore & release one permit
- Final thread waits on 3 permits

```
Thread t5 = new Thread(() -> {
    s456.acquire(3);
    System.out.println("T5 started");
    Thread.sleep((int)(100 * Math.random()));
    System.out.println("T5 finished");
});
```



Conclusion

- has nice thread facilities!
- Handling of threads through pools/queues for easy recycling and resource managament.
- Handles locking on a per-object basis natively, easy to adapt.
- Built-in Semaphores for access control and thread orchestration.


Conclusion

- has nice thread facilities!
- Handling of threads through pools/queues for easy recycling and resource managament.
- Handles locking on a per-object basis natively, easy to adapt.
- Built-in Semaphores for access control and thread orchestration.

Conclusion

- has nice thread facilities!
- Handling of threads through pools/queues for easy recycling and resource managament.
- Handles locking on a per-object basis natively, easy to adapt.
- Built-in Semaphores for access control and thread orchestration.

Conclusion

- has nice thread facilities!
- Handling of threads through pools/queues for easy recycling and resource managament.
- Handles locking on a per-object basis natively, easy to adapt.
- Built-in Semaphores for access control and thread orchestration.

Conclusion

- has nice thread facilities!
- Handling of threads through pools/queues for easy recycling and resource managament.
- Handles locking on a per-object basis natively, easy to adapt.
- Built-in Semaphores for access control and thread orchestration.

Conclusion

- has nice thread facilities!
- Handling of threads through pools/queues for easy recycling and resource managament.
- Handles locking on a per-object basis natively, easy to adapt.
- Built-in Semaphores for access control and thread orchestration.

Conclusion

- has nice thread facilities!
- Handling of threads through pools/queues for easy recycling and resource managament.
- Handles locking on a per-object basis natively, easy to adapt.
- Built-in Semaphores for access control and thread orchestration.

Conclusion

- has nice thread facilities!
- Handling of threads through pools/queues for easy recycling and resource managament.
- Handles locking on a per-object basis natively, easy to adapt.
- Built-in Semaphores for access control and thread orchestration.

Feedback

Feedback

Bibliography I

Bibliography I