Object-Oriented Programming in Java

Lecture 10 - Parallel Computing

Emily Lucia Antosch

HAW Hamburg

16.08.2025

Inhaltsverzeichnis

1. Introduction

1.1 Where are we now?

- In the last lecture, we dealt with output and input
- You can now
  - send and format output to the console in the right channel,
  - request input from the user
  - and read files in Java.
- Today we continue with **parallel computing**.

1.1 Where are we now?

1. Imperative Konzepte
2. Klassen und Objekte
3. Klassenbibliothek
4. Vererbung
5. Schnittstellen
6. Graphical User Interfaces
7. Exception Handling
8. Input and Output
9. **Multithreading (Parallel Computing)**

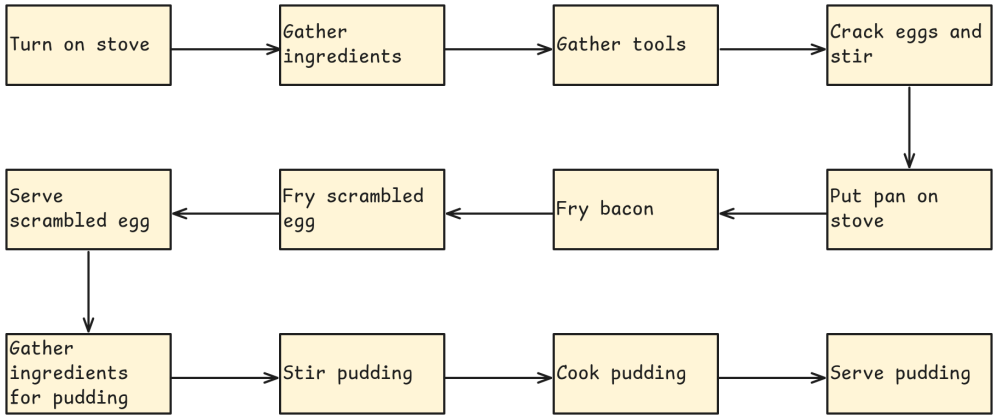
1.2 The goal of this chapter

- You execute program code simultaneously in concurrent execution threads (threads).
- You modify the states of active threads to generate the required functionality.
- You synchronize threads and objects to prevent erroneous data states due to incorrect execution orders.

2. Parallel processing

2.1 Scrambled eggs and pudding

- You make scrambled eggs and pudding.
- Possible sequence:

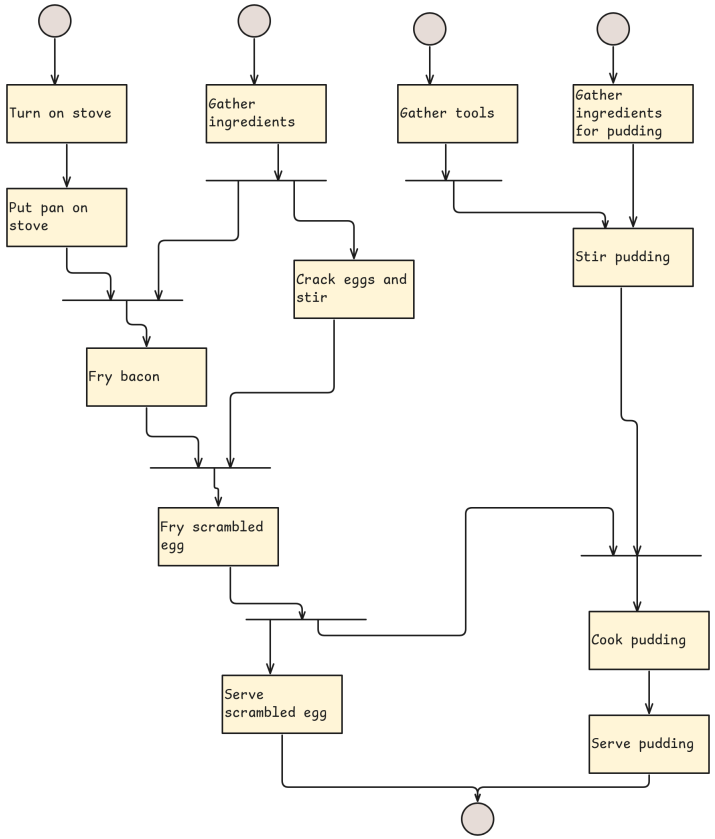


? Frage

- What could the sequence look like if four of you are cooking?
- Constraint: There is only one stove burner

2.1 Scrambled eggs and pudding

- Possible sequence
- Resource conflict: stove burner



2.1 Scrambled eggs and pudding

- Task is divided into subtasks that can be executed in parallel
- Results of subtasks must be exchanged
- Problems:
  - Dependencies: Subtasks need results of other subtasks
  - Resource conflict: Subtasks need the same resource
  - Communication overhead: Exchange of results requires resources and time
- Tasks cannot be parallelized arbitrarily or automatically.

2.1 Scrambled eggs and pudding

- Terms:
  - Thread (English for „thread“): Execution thread within a program
  - Multithreading: Multiple (parallel) execution threads within a program
- Memory:
  - Threads share the memory area of the program:
  - Therefore share variables and objects
  - Can communicate efficiently (but unsafely!) via variables and objects
- But: Each thread has its own call stack of called methods

2.1 Scrambled eggs and pudding

? Frage

- Small riddle in between:
- We have already learned about at least one parallel thread. Which one?

•

•

2.1 Scrambled eggs and pudding

? Frage

- Small riddle in between:
- We have already learned about at least one parallel thread. Which one?

- Answer:
  - Garbage Collector (free memory of unreferenced objects)
- Note:
  - Java programs create a main thread on startup
  - Set main() as the bottom method on the call stack
  - If needed, additionally a thread for the Garbage Collector is started
  - Program terminates as soon as the last associated thread has terminated

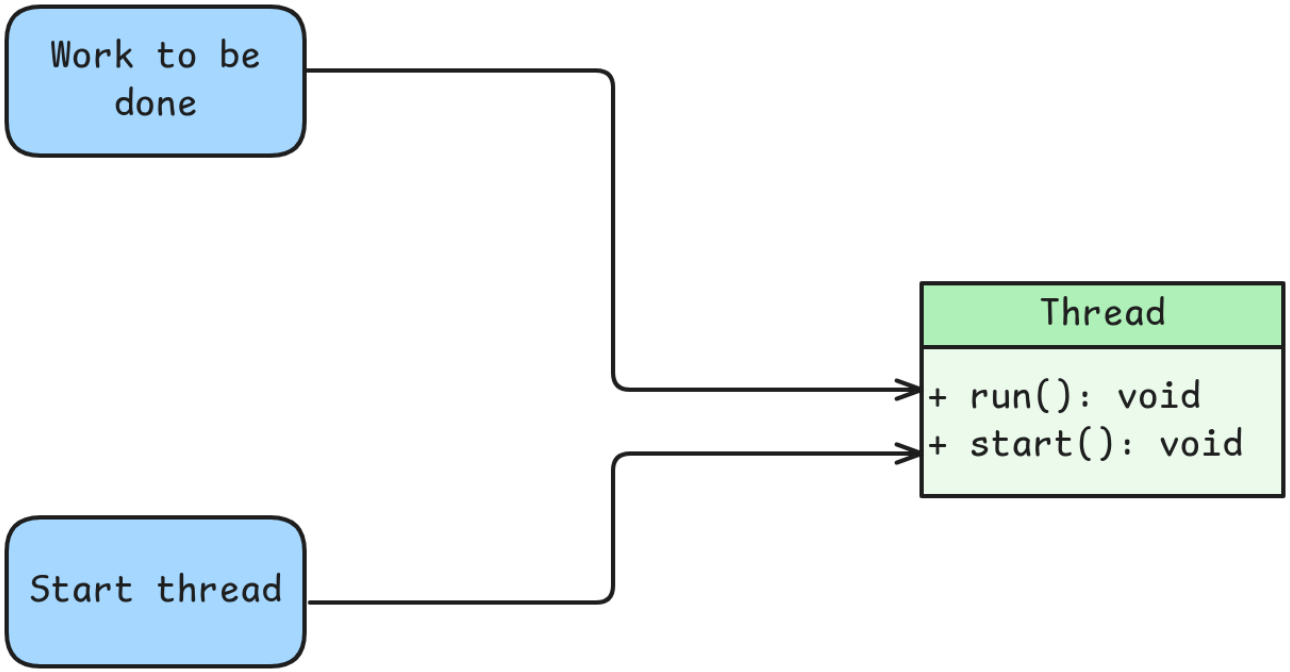
2.1 Scrambled eggs and pudding

- Allocates computing time (i.e. processors or processor cores) to programs and threads
- Waiting times of other threads or programs are used
- Pseudo-parallelism:
  - If there are more parallel execution threads than processors or processor cores
  - Scheduler distributes computing time in slices:
    - Execution in temporal alternation
    - Impression that things are processed in parallel

3. Class-based threads

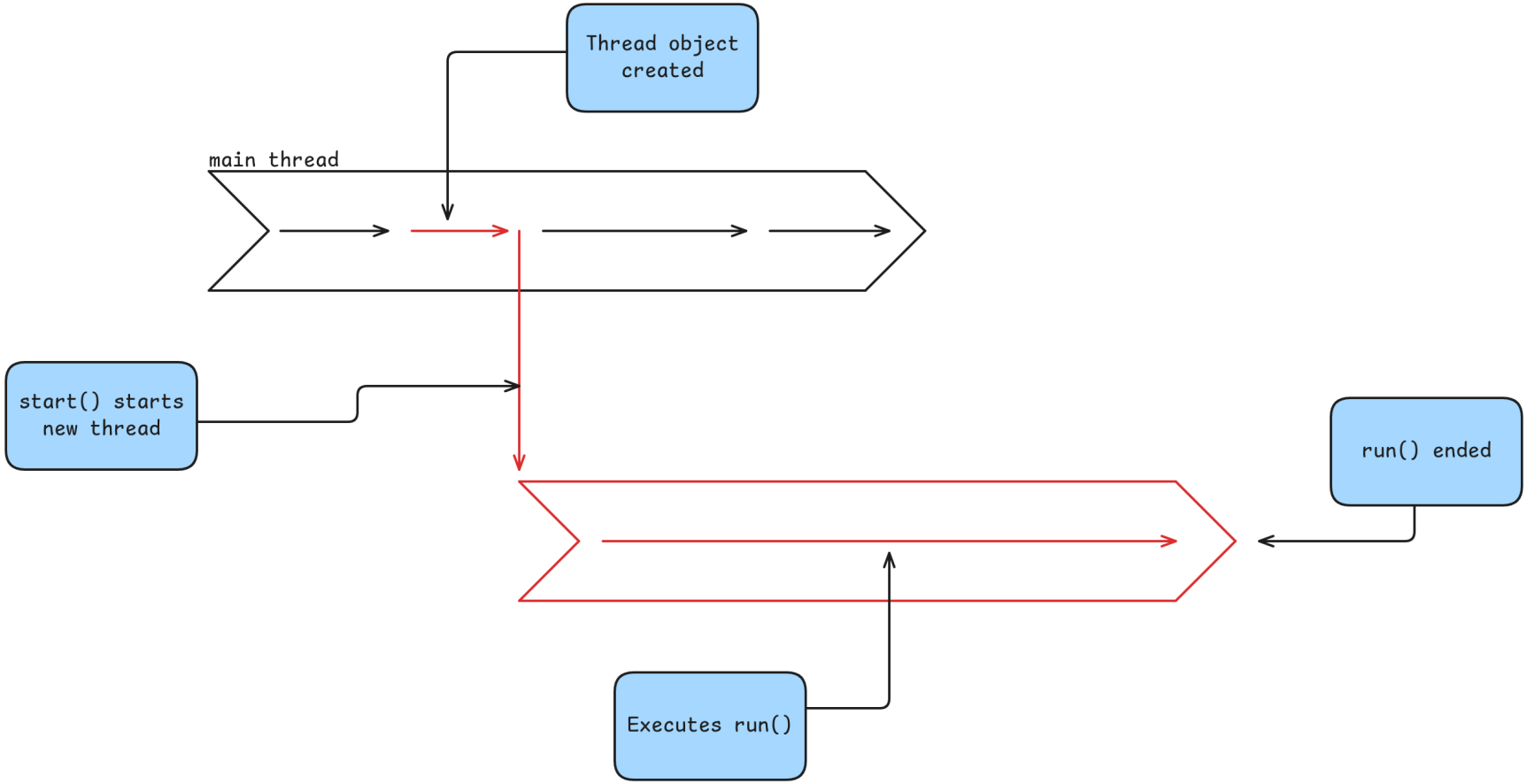
3.1 Class Thread

- Threads are created by objects of the Thread class:
- Method `start()` creates and starts parallel execution thread
- Method `run()` contains code to be executed in execution thread
- Execution thread is terminated as soon as `run()` is terminated



3.1 Class Thread

- Illustration



3.1 Class Thread

Aufgabe 1

- Let's implement this:
- Write a program that creates an additional thread.

Java

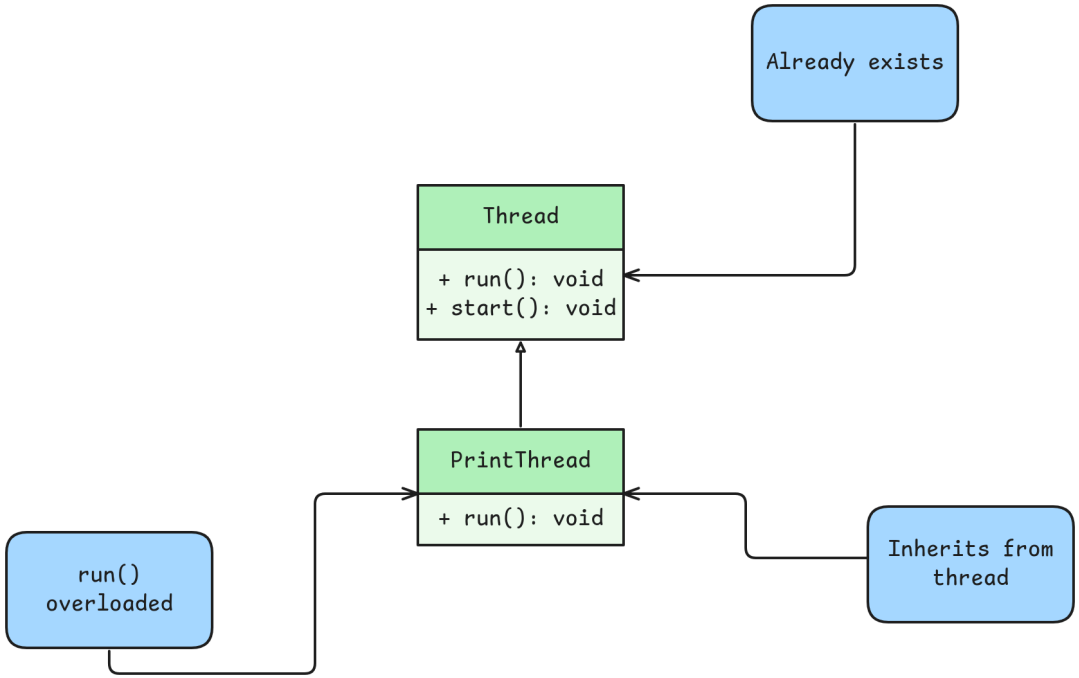
```java
public class RunThread1 {
    public static void main(String[] args) {
        Thread thread = new Thread();
        System.out.println("Object created");
        thread.start();
        System.out.println("Thread started");
    }
}
```

Frage

- But you can't see anything from the thread!
  - The run() method of the Thread class is „empty".
  - How can we make the thread output text to the console?

3.1 Class Thread

- Approach:
  - The actual work takes place in the run() method.
  - The run() method of the Thread class is „empty“.
- Derive your own Thread class from Thread and override run()



3.1 Class Thread

Aufgabe 2

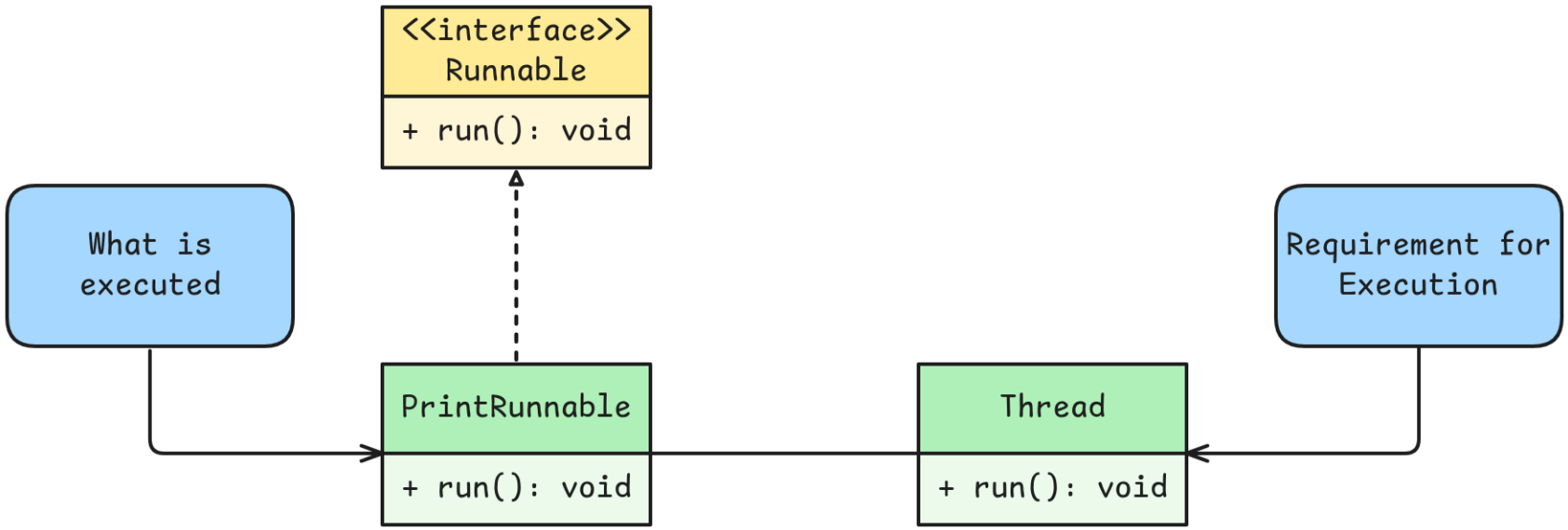
- Generate console output in an additional thread.

```java
public class PrintThread extends Thread {
    public void run() {
        System.out.println("Hooray, I'm running in parallel!");
    }
}

public class RunThread2 {
    public static void main(String[] args) {
        PrintThread thread = new PrintThread();
        System.out.println("Object created");
        thread.start();
        System.out.println("Thread started");
    }
}
```

4. Interface-based threads

4.1 Interface Runnable

- Alternative to deriving from Thread:
  - Own class implements interface `Runnable` with `run()` method
  - Runnable object is passed to Thread object: No inheritance required
- Responsibilities:
  - Runnable object contains what should be executed
  - Thread object contains everything needed for concurrency



4.1 Interface Runnable

Java

```java
public class PrintRunnable implements Runnable {
    public void run() {
        System.out.println("Hooray, I'm running in parallel!");
    }
}

public class InterfaceBased {
    public static void main(String[] args) {
        PrintRunnable runnable = new PrintRunnable();
        Thread thread = new Thread(runnable);

        System.out.println("Objects created");
        thread.start();
        System.out.println("Thread started");
    }
}
```

4.1 Interface Runnable

? Frage

- What is output?

Java

```java
class CounterRunnable implements Runnable {
    private int counter;
    public void run() {
        while (counter < 10)
            System.out.println("\t\t\tThread counter: " + counter++);
        System.out.println("\t\t\tExiting run()");
    }
}
public class Counters {
    private static int counter;
    public static void main(String[] args) {
        new Thread(new CounterRunnable()).start();
        while (counter < 10)
            System.out.println("Main counter: " + counter++);
        System.out.println("Exiting main()");
    }
}
```

4.1 Interface Runnable

- Methods `run()` and `main()` count to 9
- Unpredictable who finishes first
- Example output (right):
  - `main` thread finished first
  - Thread with `run()` continues running

```
Main counter: 0
                    Thread counter: 0
Main counter: 1
                    Thread counter: 1
Main counter: 2
                    Thread counter: 2
Main counter: 3
                    Thread counter: 3
Main counter: 4
Main counter: 5
Main counter: 6
                    Thread counter: 4
Main counter: 7
Main counter: 8
Main counter: 9
Exiting main()
                    Thread counter: 5
                    Thread counter: 6
                    Thread counter: 7
                    Thread counter: 8
                    Thread counter: 9
                    Exiting run()
```

main() beendet

5. States and selected methods

5.1 Thread states

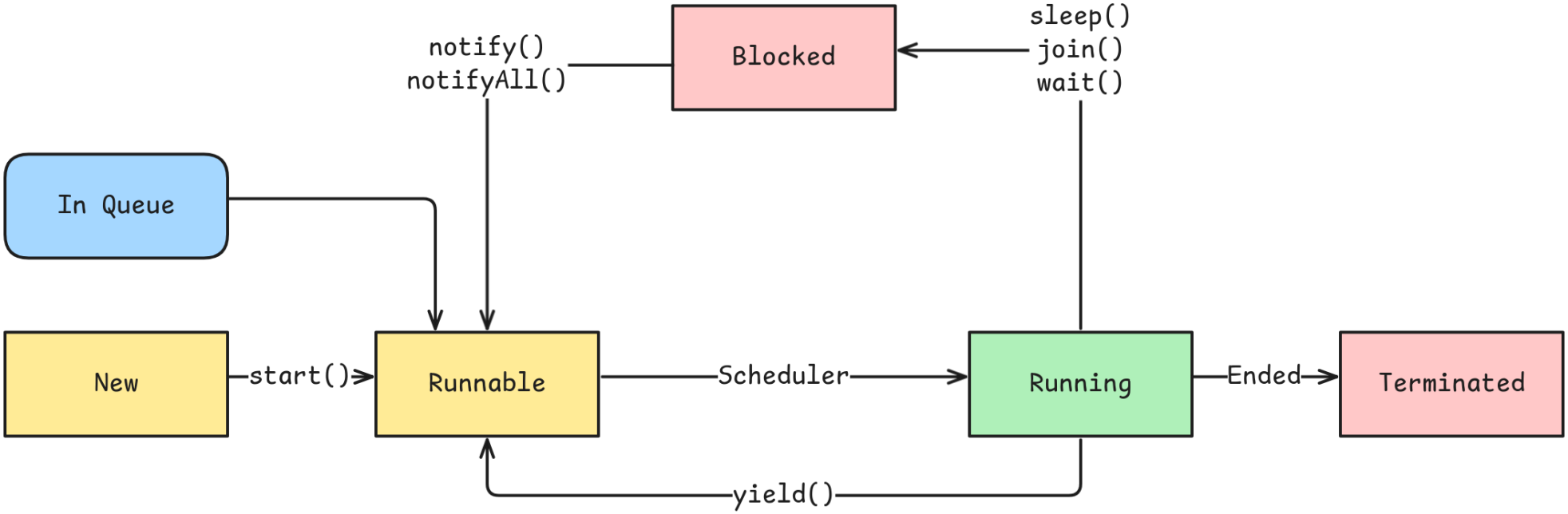
? Frage

- Imagine you were a thread:
  - What states could you reasonably take?
  - What state transitions would make sense?

- Don’t forget the following:
  - What happens when there are more threads than processors?
  - What should you do when waiting for input?

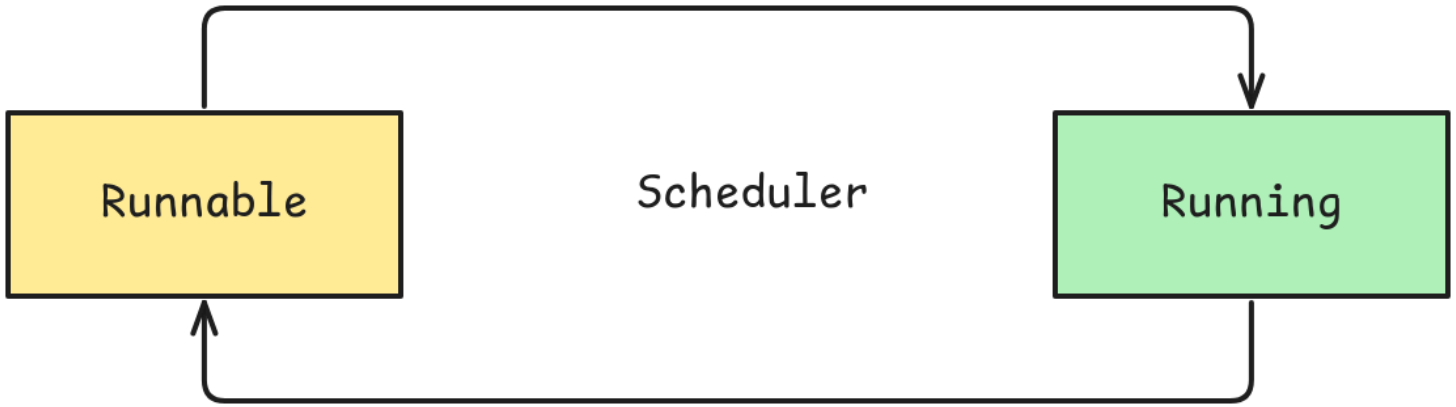
5.1 Thread states

- New: Java object created, but not yet started as a thread
- Runnable: Ready to be executed. Waiting for processor.
- Running: Has processor and is currently being executed
- Blocked: Is not executed and would not be even with a free processor
- Terminated: Thread terminated. Java object still exists!



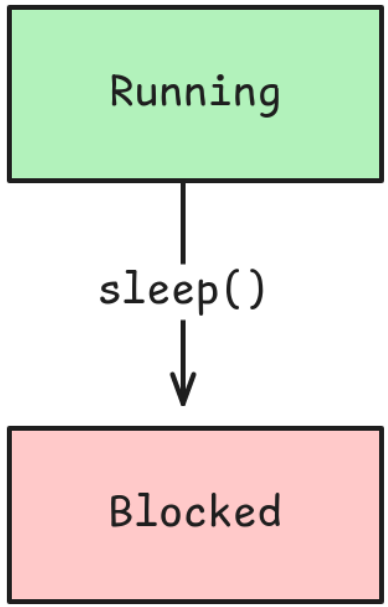
5.1 Thread states

- Assigns computing time to threads (i.e. Runnable becomes Running)
- Withdraws processor from threads again (i.e. Running becomes Runnable):
  - Required if more threads than processors exist
  - Idea: Threads receive computing time alternately
- Control of behavior:
  - Scheduler is not controllable
  - No guarantee that threads receive computing time alternately
  - `setPriority()` sets priority, but no guarantee how scheduler considers it
  - „The scheduler is a diva!“



5.1 Thread states

- Put running thread in blocked state for a certain time
- Pass waiting time in milliseconds as parameter (data type long)
- Early wake-up:
  - Thread can be „woken up“ prematurely by interrupt() method
  - Throws exception of type InterruptedException




```java
MyThread thread = new MyThread();
thread.start();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

5.1 Thread states

- Make the window blink (every 0.75 s alternating between yellow and light gray):

Java

```java
public class FlashLight {
    private boolean isLightOn;
    private JFrame frame;
    private FlashLight() {
        frame = new JFrame("Flashing light");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 250);
        frame.setVisible(true);
    }
    public void switchLight() {
        isLightOn = !isLightOn;
        if (isLightOn)
            frame.getContentPane().setBackground(Color.YELLOW);
        else
            frame.getContentPane().setBackground(Color.LIGHT_GRAY);
    }
    public static void main(String[] args) {
        FlashLight flashLight = new FlashLight();
    }
}
```

5.1 Thread states

- Required for blinking:
  - Thread that calls the switchLight() method every 0.75 s

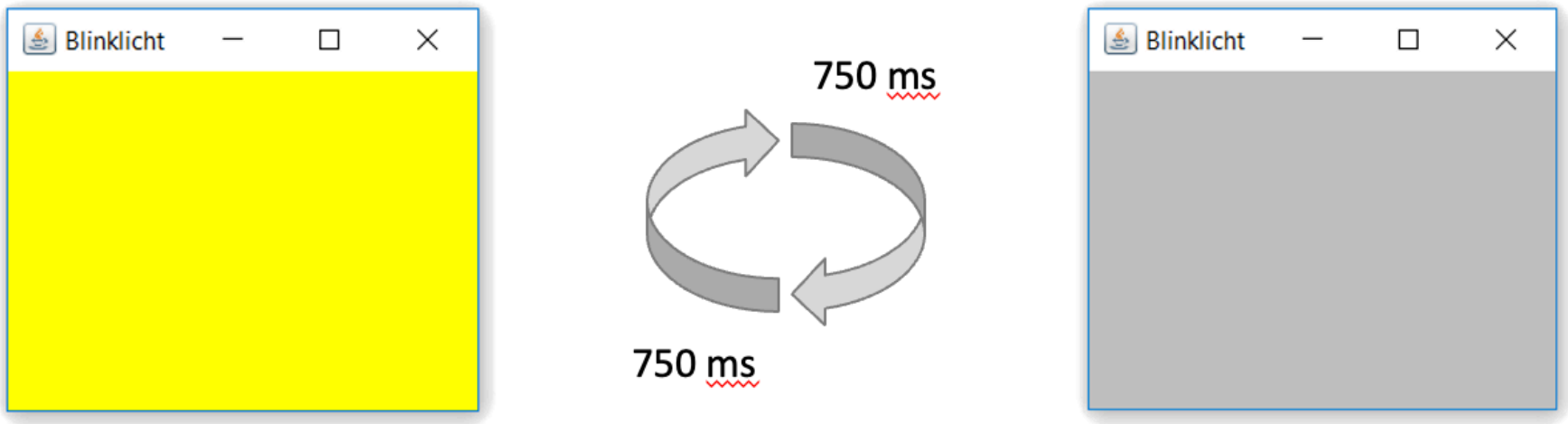
Java

```java
class FlashThread extends Thread {
    private FlashLight flashLight;

    public FlashThread(FlashLight flashLight) {
        this.flashLight = flashLight;
    }

    public void run() {
        while (true) {
            flashLight.switchLight();
            try {
                Thread.sleep(750);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

5.1 Thread states

- Creation and starting of the thread in FlashLight:

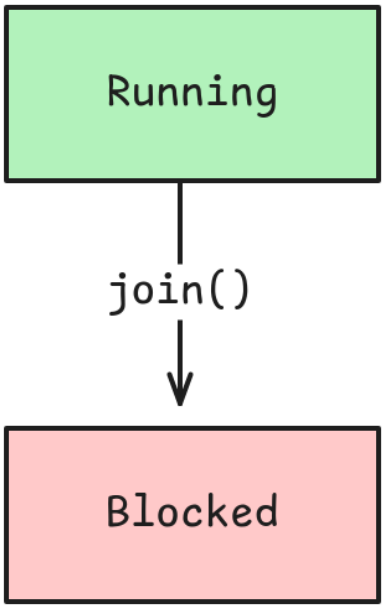
```java
public static void main(String[] args) {
    FlashLight flashLight = new FlashLight();
    FlashThread thread = new FlashThread(flashLight);
    thread.start();
}
```



5.1 Thread states

- Make running thread wait for the end of another thread
- Example:
  - Waits at `thread.join()` until `thread` terminates
  - Only then console output occurs

```java
public static void main(String[] args) {
    MyThread thread = new MyThread();
    thread.start();
    thread.join();
    System.out.println("We have joined!");
}
```



- Maximum waiting time:
  - Maximum waiting time can be specified as parameter (data type `long`)
  - What is this needed for? (After all, you don't wait for no reason)

5.1 Thread states

? Frage

- What does this thread do?

```java
public class SleepyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("I'm sooo tired ...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Okay, I'm awake again.");
    }
}
```

5.1 Thread states

- It continues:
  - What output is produced?
  - What output would be produced without the line sleepy.join()?
  - What output would be produced with sleepy.join(1500)?

Java

```java
public class JoinThreads {
    public static void main(String[] args) throws InterruptedException {
        SleepyThread sleepy = new SleepyThread();
        sleepy.start();

        while (sleepy.isAlive()) {
            System.out.println("Wake up!");
            Thread.sleep(400);
            sleepy.join();
        }
        System.out.println("At last ...");
    }
}
```

6. Synchronization

6.1 Synchronization

- Class represents a bank account with methods for deposits and withdrawals
- Account movements in parallel via threads (e.g. ATM, counter, direct debit)

```java
public class Account {
    private double balance;

    public void deposit(double amount) {
        double newBalance = balance + amount;
        if (newBalance > balance)
            balance = newBalance;
    }

    public void withdraw(double amount) {
        double newBalance = balance - amount;
        if (newBalance >= 0.0)
            balance = newBalance;
    }
}
```

- What happened here?!
  - You withdraw 50 € while 50 € is credited as a transfer.
  - Afterwards there are 50 € less than before in the account.

6.1 Synchronization

- Cause:
  - Threads simultaneously execute methods deposit() and withdraw()
  - Both methods access variable balance.

Thread 1: *deposit()*	Betrag	Thread 2: *withdraw()*	Betrag	*balance*
`newBalance = balance + amount;`	(5050)			5000
		`newBalance = balance - amount;`	(4950)	5000
`if (newBalance > balance)` `  balance = newBalance;`				5050
		`if (newBalance >= 0.0)` `  balance = newBalance;`		4950

6.1 Synchronization

- Two threads share a variable.
  - Race Condition: Result of the program depends on access order
- When does the result depend on which thread is „faster“?
  - Both threads read the variable
  - One thread reads, one thread writes to the variable
  - Both threads write to the variable
- Answer:
  - Race condition when at least one thread writes

Thread 1	Thread 2	Race Condition
`read`	`read`	None, both threads read the same data
`read`	`write`	Thread 1 may read value before *or* after thread 2 writes
`write`	`write`	Last-written value remains

Tabelle 1: Formats and Flags

6.1 Synchronization

- Keyword `synchronized` for methods:
  - Object is locked as soon as a thread enters a synchronized method
  - Object is released again when thread leaves the method
- Synchronized methods (mutual exclusion):
  - Object locked: Threads cannot enter synchronized methods. (All synchronized methods are locked, not just the one currently being executed!)
  - Threads wait in blocked state until the object is released again.
- Non-synchronized methods:
  - Threads can enter non-synchronized methods when object is locked.

Aufgabe 3

- Help your bank:
  - Ensure that nothing goes wrong with deposits and withdrawals.

6.1 Synchronization

Synchronization via `synchronized`:

Java

```java
public class Account {
    private double balance;

    public synchronized void deposit(double amount) {
        double newBalance = balance + amount;
        if (newBalance > balance)
            balance = newBalance;
    }

    public synchronized void withdraw(double amount) {
        double newBalance = balance - amount;
        if (newBalance >= 0.0)
            balance = newBalance;
    }
}
```

7. License Notice

7.1 Attribution

- This work is shared under the CC BY-NC-SA 4.0 License and the respective Public License
- link(„https://creativecommons.org/licenses/by-nc-sa/4.0/“)
- This work is based off of the work Prof. Dr. Marc Hensel.
- Some of the images and texts, as well as the layout were changed.
- The base material was supplied in private, therefore the link to the source cannot be shared with the audience.