CARL VON OSSIETZKY universität OLDENBURG

Sommersemester 2020

Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

Java-Threads und Parallele Programmierung I

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Parallelität und Nebenläufigkeit
- Threads in Java
- Kommunikation zwischen Threads
- Synchronisation

CARL VON OSSIETZKY universität OLDENBURG

Parallelisierung

- Garbage Collector
- GUIs
- Multimedia
- Web-Anwendungen
- Client-Server Anwendungen
- Zeitgesteuerte Anwendungen
- Computerspiele
- Simulationen
- Laufzeitverbesserung

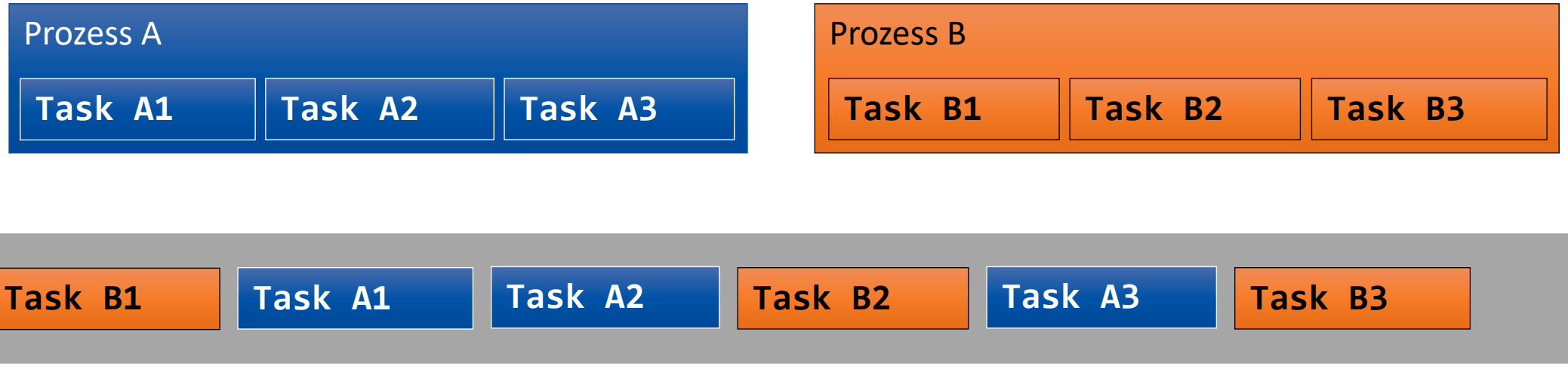
CARL VON OSSIETZKY universität OLDENBURG

Begriffe

- Programm: Java-Code und alle verwendeten Klassen, Bibliotheken und mit dem Code ausgelieferten Ressourcen
- Prozess: Ausführung eines Programms auf einem Prozessor (CPU)
- Thread: Teilprozess, der sich mit anderen Threads des gleichen Prozesses den Speicher teilt

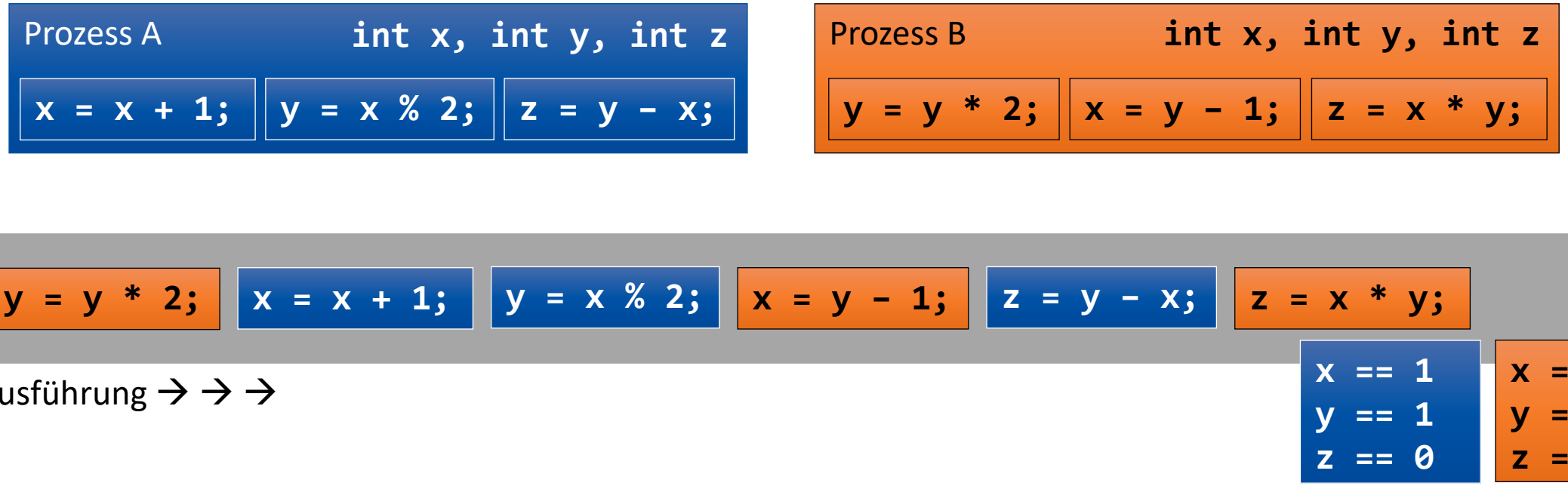
Multitasking

- Ein Prozess oder ein Thread wird in einzelne Arbeitsschritte (Tasks) unterteilt
- Die Prozesssteuerung (Betriebssystem, JVM, …) schaltet so schnell zwischen den Tasks um, dass es wie gleichzeitiges Ablaufen wirkt (pseudo-parallel)



Multitasking

- Ein Prozess oder ein Thread wird in einzelne Arbeitsschritte (Tasks) unterteilt
- Die Prozesssteuerung (Betriebssystem, JVM, …) schaltet so schnell zwischen den Tasks um, dass es wie gleichzeitiges Ablaufen wirkt (pseudo-parallel)



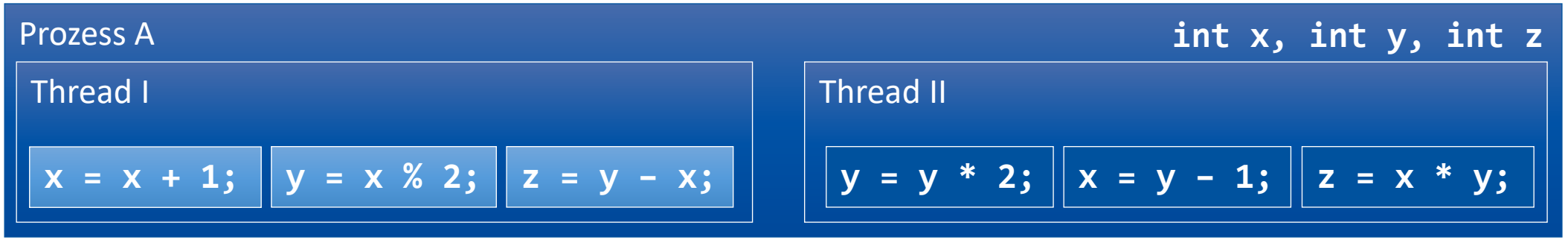
Multitasking (2)

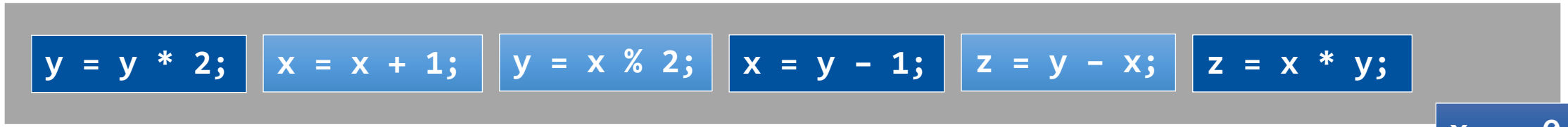
- Ein Prozess oder ein Thread wird in einzelne Arbeitsschritte (Tasks) unterteilt
- Die Prozesssteuerung (Betriebssystem, JVM, …) schaltet so schnell zwischen den Tasks um, dass es wie gleichzeitiges Ablaufen wirkt (pseudo-parallel)

Ausführung → → →

Multitasking (2)

- Ein Prozess oder ein Thread wird in einzelne Arbeitsschritte (Tasks) unterteilt
- Die Prozesssteuerung (Betriebssystem, JVM, …) schaltet so schnell zwischen den Tasks um, dass es wie gleichzeitiges Ablaufen wirkt (pseudo-parallel)



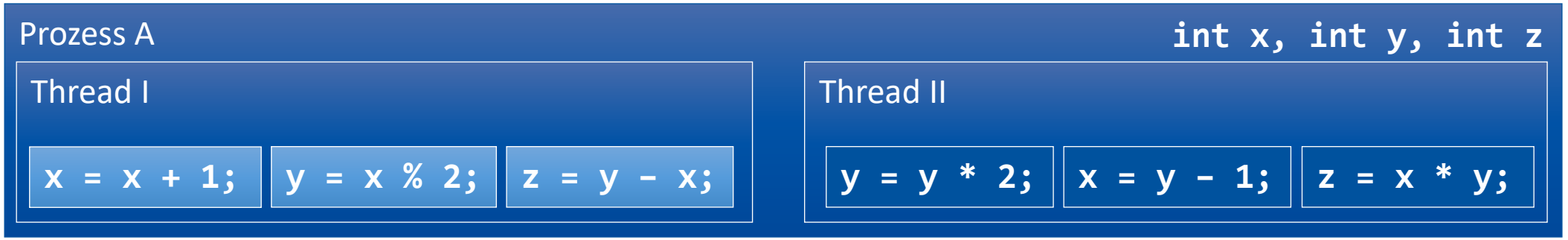


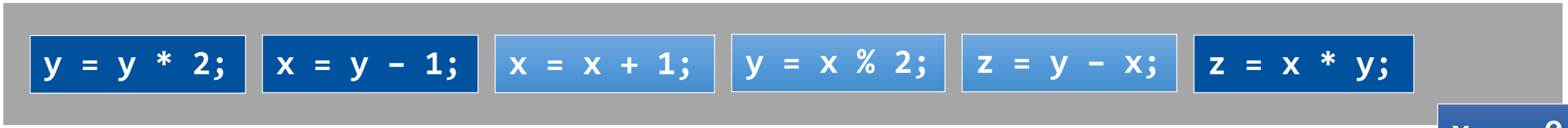
Ausführung → → →

```
x == 0
y == 1
z == 0
```

Multitasking (2)

- Ein Prozess oder ein Thread wird in einzelne Arbeitsschritte (Tasks) unterteilt
- Die Prozesssteuerung (Betriebssystem, JVM, …) schaltet so schnell zwischen den Tasks um, dass es wie gleichzeitiges Ablaufen wirkt (pseudo-parallel)





Ausführung → → →

```
x == 0
y == 0
z == 0
```

CARL VON OSSIETZKY universität OLDENBURG

Scheduling in Java

- Präemptives Scheduling
- Keine festgelegte Aktivitätszeitspanne
- Keine festgelegte Aktivierungsreihenfolge
- Keine Fairness
  → ein Thread muss ggf. mehrfach warten, während ein anderer mehrfach aktiviert wird

- Fazit: nicht auf Scheduling-Reihenfolge verlassen!

Multitasking vs. Multiprocessing

Multitasking

- Multitasking von Prozessen: Betriebssystem führt mehrere Prozesse pseudo-parallel aus
- Multitasking von Threads: JVM führt mehrere Threads pseudo-parallel aus (Multithreading)
  → Nebenläufigkeit

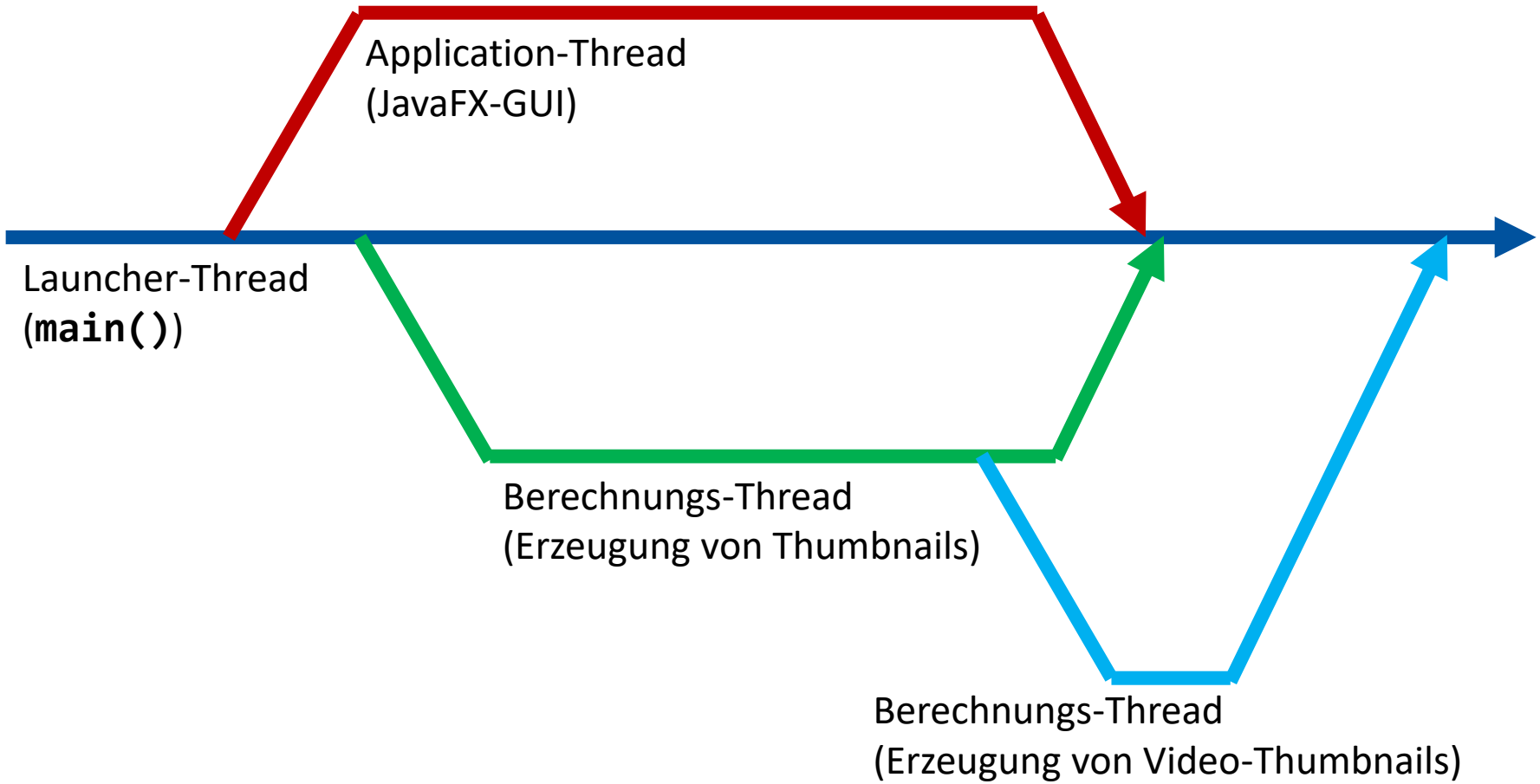
Multiprocessing

- Voraussetzung: System mit mehreren Prozessoren (CPUs) oder mehreren Kernen in einem Prozessor oder beidem
- Verschiedene Prozesse, die unterschiedlichen Prozessoren zugeordnet sind, können gleichzeitig ausgeführt werden
  → echte Parallelität

Threads: Beispiel

Programm zur Anzeige von Bildern: **`ImageViewer.java`**

Ausführung von **`ImageViewer.java`**:
Prozess, bestehend aus mehreren pseudo-parallelen Threads.



CARL VON OSSIETZKY universität OLDENBURG

Threads: Java

```java
public class MessageThread extends Thread {
    private String message;

    public MessageThread(String msg) {
        message = msg;
    }

    @Override
    public void run() {
        while (true) {
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        MessageThread hello = new MessageThread("Hello");
        MessageThread world = new MessageThread("World");
        hello.start();
        world.start();
    }
}
```

```
Hello
Hello
Hello
Hello
World
World
World
World
World
Hello
Hello
Hello
…
```

Threads in Java

- Klasse ableiten von **`Thread`**
- **`run()`**-Methode implementieren
- Klasse instanziieren
- **`start()`**-Methode der Instanz aufrufen
  - ➢ **`run()`**-Methode wird nebenläufig ausgeführt
    - direkter Aufruf von **`run()`** wird *nicht* nebenläufig ausgeführt!

- Was ist, wenn die neue Klasse noch von etwas anderem erben soll als von **`Thread`**?
  → funktionales Interface **`Runnable`**

CARL VON OSSIETZKY universität OLDENBURG

Threads: Java (2)

```
public class MessageRunner implements Runnable {
    private String message;

    public MessageRunner(String msg) {
        message = msg;
    }

    @Override
    public void run() {
        while (true) {
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        MessageRunner hello = new MessageRunner("Hello");
        MessageRunner world = new MessageRunner("World");
        new Thread(hello).start();
        new Thread(world).start();
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Threads: Java (3)

```
new Thread(() -> { while (true) System.out.println("Hello"); }).start();
new Thread(() -> { while (true) System.out.println("World"); }).start();
```

CARL VON OSSIETZKY universität OLDENBURG

Lebenszyklus

- Ein Thread wird beendet, wenn seine **`run()`**-Methode beendet ist
- Eine Programmausführung wird beendet, wenn alle (regulären) Threads einschließlich des **`main()`**-Threads beendet sind
  - es gibt besondere Hintergrund-Threads (Daemon-Threads), die hierfür nicht zählen
- Jedem Thread ist ein **`Thread`**-Objekt zugeordnet
  - speichert den Zustand des Threads
  - kann über die zur Verfügung gestellten Methoden manipuliert werden

Thread-Methoden

- ~~**`destroy()`**~~, ~~**`resume()`**~~, ~~**`stop()`**~~, ~~**`suspend()`**~~: diese Methoden sind unsicher (*inherently unsafe*) und können leicht zu schweren Fehlern führen
  - ➢sie dürfen ***nicht*** verwendet werden!
- **`sleep(millis: long)`**: Pausiert die Ausführung des *aktuellen* Threads für die angegebene Länge in Millisekunden
- **`join()`**: Wartet (blockiert) bis der Thread beendet wird
- **`join(millis: long)`**: Wartet maximal für die angegebene Länge in Millisekunden
- **`sleep()`** und **`join()`** können eine **`InterruptedException`** werfen → gleich

CARL VON OSSIETZKY universität OLDENBURG

Thread-Methoden: Beispiel

```
public class MessageRunner implements Runnable {
    public static final int MESSAGE_COUNT = 3;
    …
    @Override
    public void run() {
        for (int i = 0; i < MESSAGE_COUNT; i++) {
            System.out.println(message);
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) { }
        }
    }

    public static void main(String[] args) {
        MessageRunner hello = new MessageRunner("Hello");
        MessageRunner world = new MessageRunner("World");
        new Thread(hello).start();
        new Thread(world).start();
        System.out.println("Done.");
    }
}
```

```
Done.
Hello
World
World
Hello
Hello
World
```

Thread-Methoden: Beispiel

```
public static void main(String[] args) {
    MessageRunner hello = new MessageRunner("Hello");
    MessageRunner world = new MessageRunner("World");
    Thread t1 = new Thread(hello);
    Thread t2 = new Thread(world);
    t1.start();
    t2.start();
    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) { }
    System.out.println("Done.");
}
```

```
Hello
World
World
Hello
World
Hello
Done.
```

CARL VON OSSIETZKY universität OLDENBURG

Thread-Methoden: Beispiel (2)

```
public class MessageRunner implements Runnable {
    public static final String[] MESSAGES = new String[] { "Tick", "Trick", "Track" };
    …

    public static void main(String[] args) {
        Thread[] threads = new Thread[MESSAGES.length];
        for (int i = 0; i < MESSAGES.length; i++) {
            threads[i] = new Thread(new MessageRunner(MESSAGES[i]));
        }
        for (int i = 0; i < MESSAGES.length; i++) {
            threads[i].start();
        }
        try {
            for (int i = 0; i < MESSAGES.length; i++) {
                threads[i].join();
            }
        } catch (InterruptedException e) { }
        System.out.println("Done.");
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Thread-Methoden: Interrupt

- **`interrupt()`**: Signalisiert, dass der Thread unterbrochen werden soll
  - Normalerweise: Setzt das **`interrupted`**-Flag des Threads auf **`true`**
  - Wenn auf dem Thread gerade **`sleep()`** oder **`join()`** aufgerufen wurde: Wirft eine **`InterruptedException`**
- **`isInterrupted()`**: Prüft, ob das **`interrupted`**-Flag des Threads gesetzt ist
- **`interrupted()`**: Prüft ebenfalls, ob das **`interrupted`**-Flag des *aktuellen* Threads gesetzt ist, *und setzt das Flag danach zurück*
- **`currentThread()`**: Gibt das aktuelle Thread-Objekt zurück

Thread-Methoden: Beispiel (3)

```
public class MessageRunner implements Runnable {

    public static final String[] MESSAGES = new String[] { "Tick", "Trick", "Track" };

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println(message);
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                break;
            }
        }
        System.out.println("No more " + message + " messages.");
    }

    …
```

Der Umweg über **Thread.currentThread** ist nötig, weil das **Runnable**-Interface keinen Zugriff auf die **Thread**-Methode **isInterrupted** hat.

Thread-Methoden: Beispiel (3)

```
    public static void main(String[] args) {
        Thread[] threads = new Thread[MESSAGES.length];
        for (int i = 0; i < MESSAGES.length; i++) {
            threads[i] = new Thread(new MessageRunner(MESSAGES[i]));
        }
        for (int i = 0; i < MESSAGES.length; i++) {
            threads[i].start();
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) { }
        for (int i = 0; i < MESSAGES.length; i++) {
            threads[i].interrupt();
        }
        System.out.println("Done.");
    }
}
```

```
Track
Tick
Trick
…
Tick
Trick
Track
Track
Tick
Trick
No more Trick messages.
No more Tick messages.
No more Track messages.
Done.
```

CARL VON OSSIETZKY universität OLDENBURG

Kommunikation zwischen Threads

- Kommunikation = Datenaustausch
- Über gemeinsame Variablen bzw. Objekte möglich, da Threads sich einen gemeinsamen Adressraum teilen
- Jedem Thread Zugriff auf gemeinsame Datenobjekte geben (z.B. an den Konstruktor übergeben)
- Anschließend gemeinsam darauf zugreifen

CARL VON OSSIETZKY universität OLDENBURG

Kommunikation: Beispiel

```
class NumberData {

    private int value;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Kommunikation: Beispiel

```
public class NumberReader implements Runnable {
    protected NumberData data;
    protected int addition;

    public NumberReader(NumberData data, int addition) {
        this.data = data;
        this.addition = addition;
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            int value = data.getValue() + addition;
            System.out.println(value);
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                break;
            }
        }
    } }
```

CARL VON OSSIETZKY universität OLDENBURG

Kommunikation: Beispiel

```
class NumberWriter extends NumberReader {

    Random random = new Random();

    public NumberWriter(NumberData data, int addition) {
        super(data, addition);
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            int value = random.nextInt(addition);
            data.setValue(value);
            System.out.println(" -> " + value);
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                break;
            }
        }
    } }
```

CARL VON OSSIETZKY universität OLDENBURG

Kommunikation: Beispiel

```
public static final int THREAD_COUNT = 4;

public static void main(String[] args) {
    NumberData data = new NumberData();
    Thread[] threads = new Thread[4];
    threads[0] = new Thread(new NumberWriter(data, 100));
    for (int i = 1; i < THREAD_COUNT; i++) {
        threads[i] = new Thread(new NumberReader(data, i));
    }
    for (int i = 0; i < THREAD_COUNT; i++) {
        threads[i].start();
    }
    try {
        Thread.sleep(200);
    } catch (InterruptedException e) { }
    for (int i = 0; i < THREAD_COUNT; i++) {
        threads[i].interrupt();
    }
}
```

Kommunikation: Beispiel

```
 -> 91
94
93
92
93
94
92
93
94
92
 -> 76
78
77
79
78
77
79
…
```

```
…
 -> 9
11
10
12
11
10
12
11
10
12
 -> 79
81
80
82
81
80
82
```

CARL VON OSSIETZKY universität OLDENBURG

Konflikte

CARL VON OSSIETZKY universität OLDENBURG

Konflikte: Beispiel

```
class Account {

    private int balance = 0;

    public void setBalance(int balance) {
        this.balance = balance;
    }

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

CARL VON OSSIETZKY universität OLDENBURG

Konflikte: Beispiel

```
public class MoneyRunner implements Runnable {

    private Account account;
    private int amount;

    public MoneyRunner(Account account, int amount) {
        this.account = account;
        this.amount = amount;
    }

    @Override
    public void run() {
        int balance = account.getBalance();
        int total = balance + amount;
        System.out.println(balance + " + " + amount + " = " + total);
        account.setBalance(total);
    }

…
```

Konflikte: Beispiel

```
…

    public static void main(String[] args) {
        Account acc = new Account();
        Thread t1 = new Thread(new MoneyRunner(acc, 5));
        Thread t2 = new Thread(new MoneyRunner(acc, 10));
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.err.println("Thread was interrupted!");
        }
        System.out.println(acc.getBalance());
    }
}
```

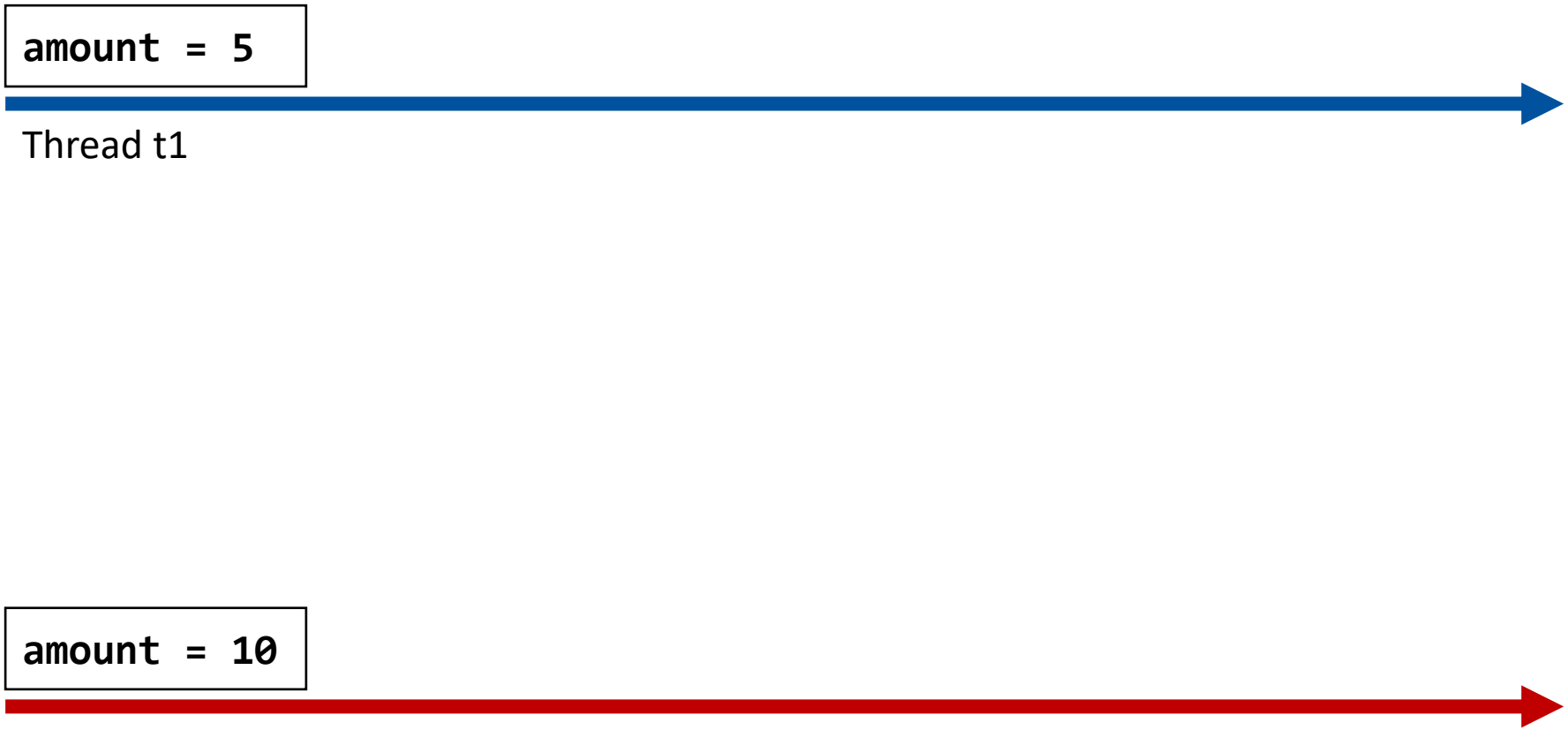
```
0 + 10 = 10
10 + 5 = 15
15
```

```
0 + 10 = 10
0 + 5 = 5
5
```

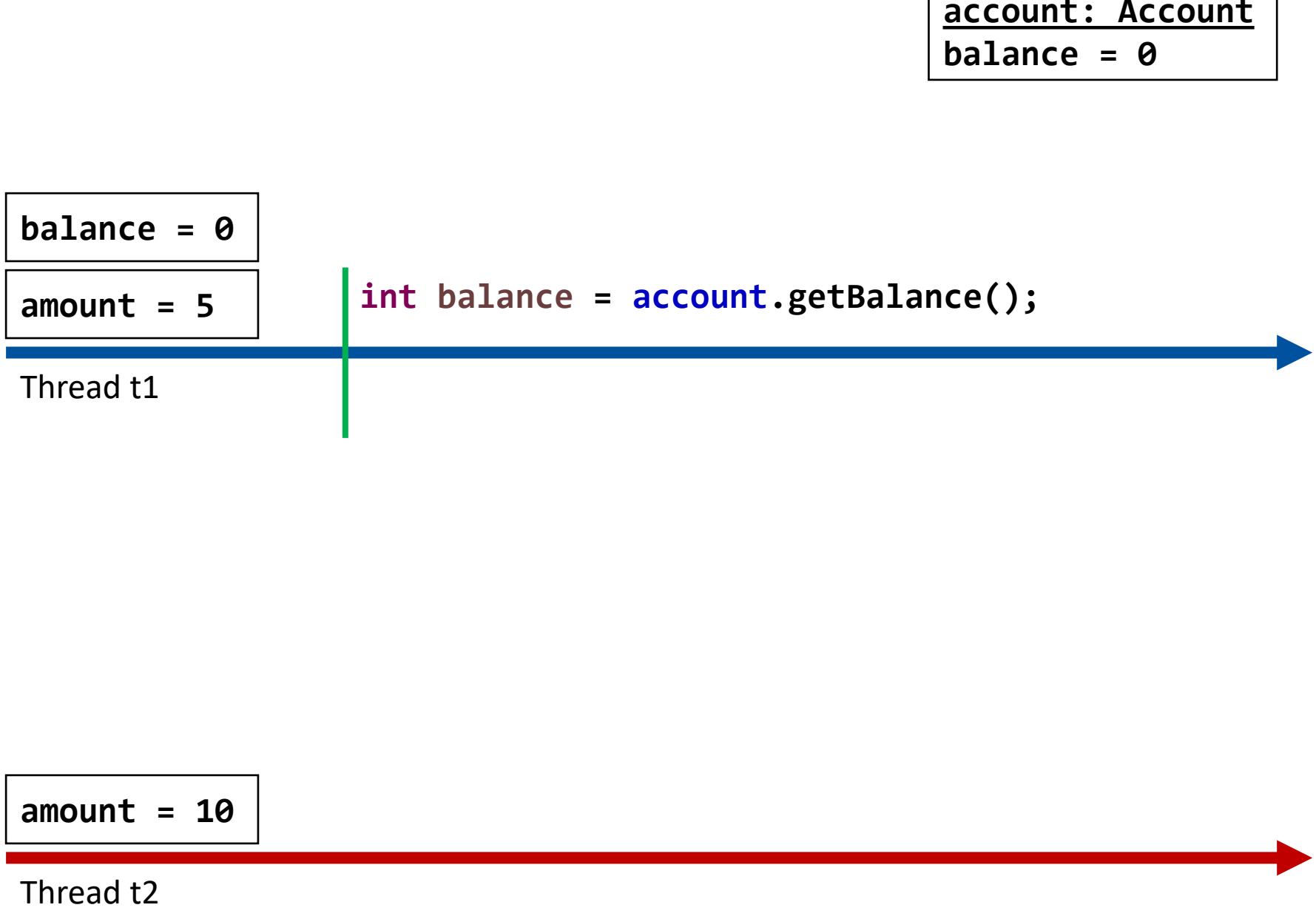
```
0 + 10 = 10
0 + 5 = 5
10
```

Konflikte: Beispiel

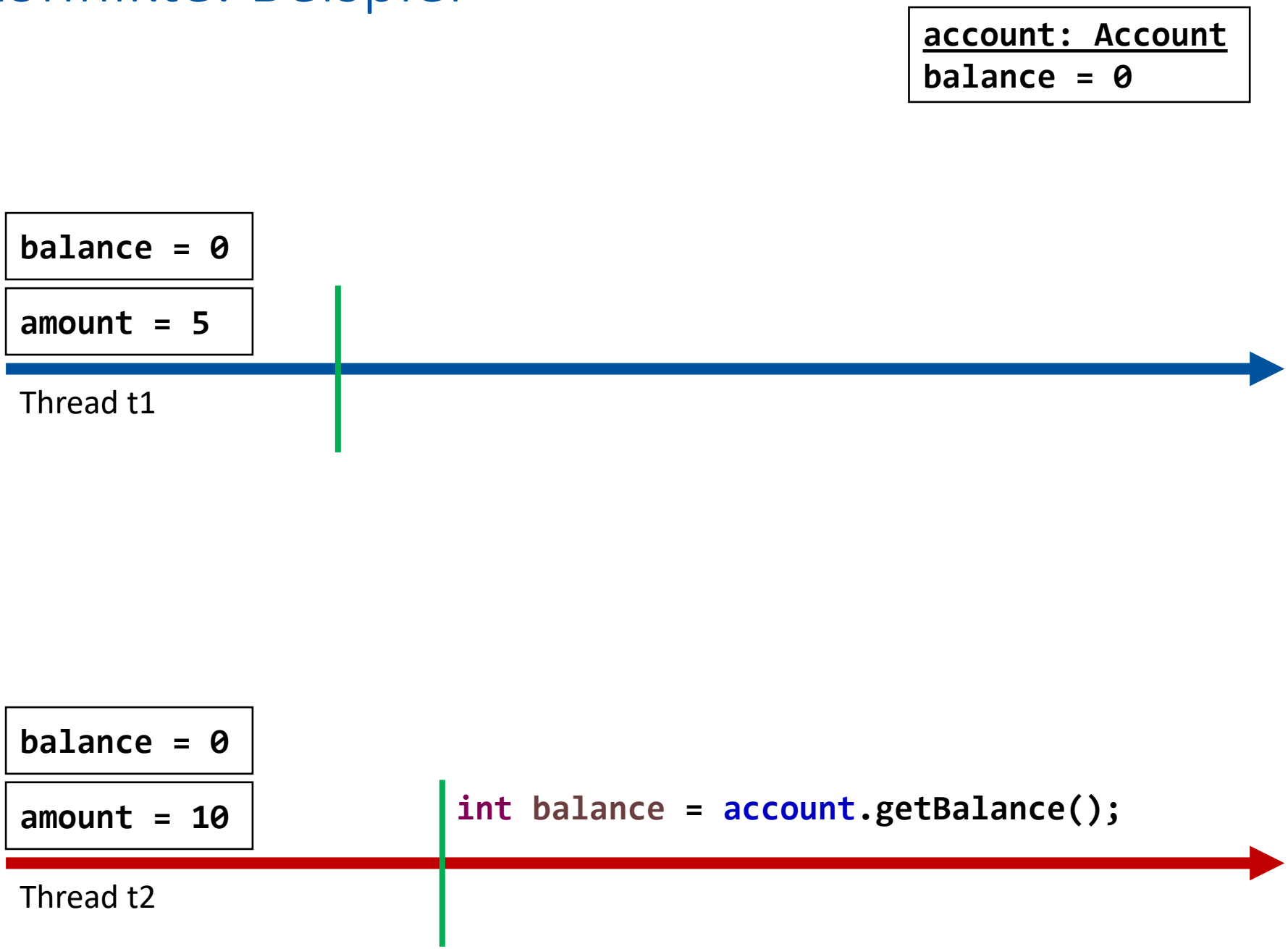
```
account: Account
balance = 0
```



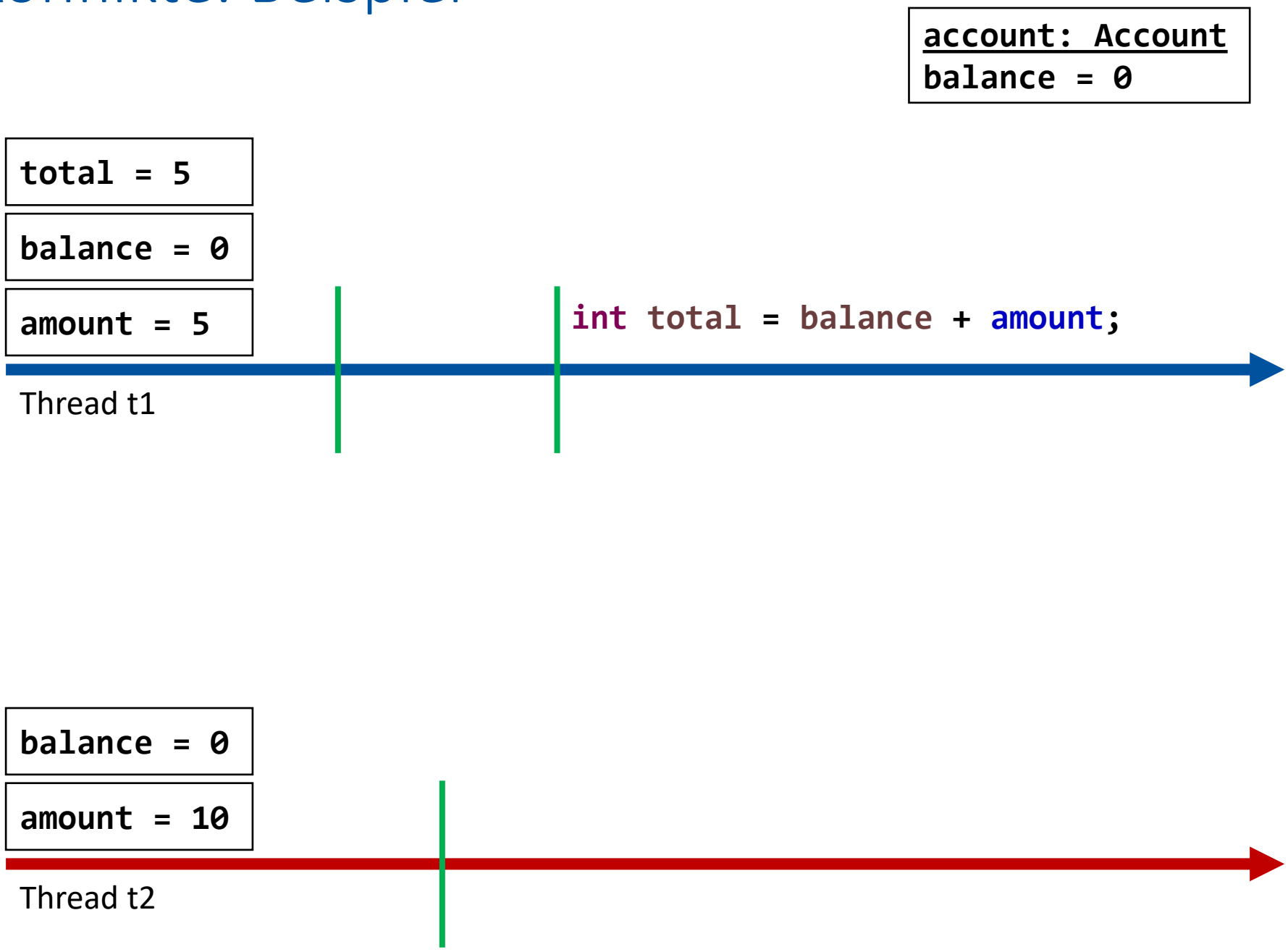
Konflikte: Beispiel



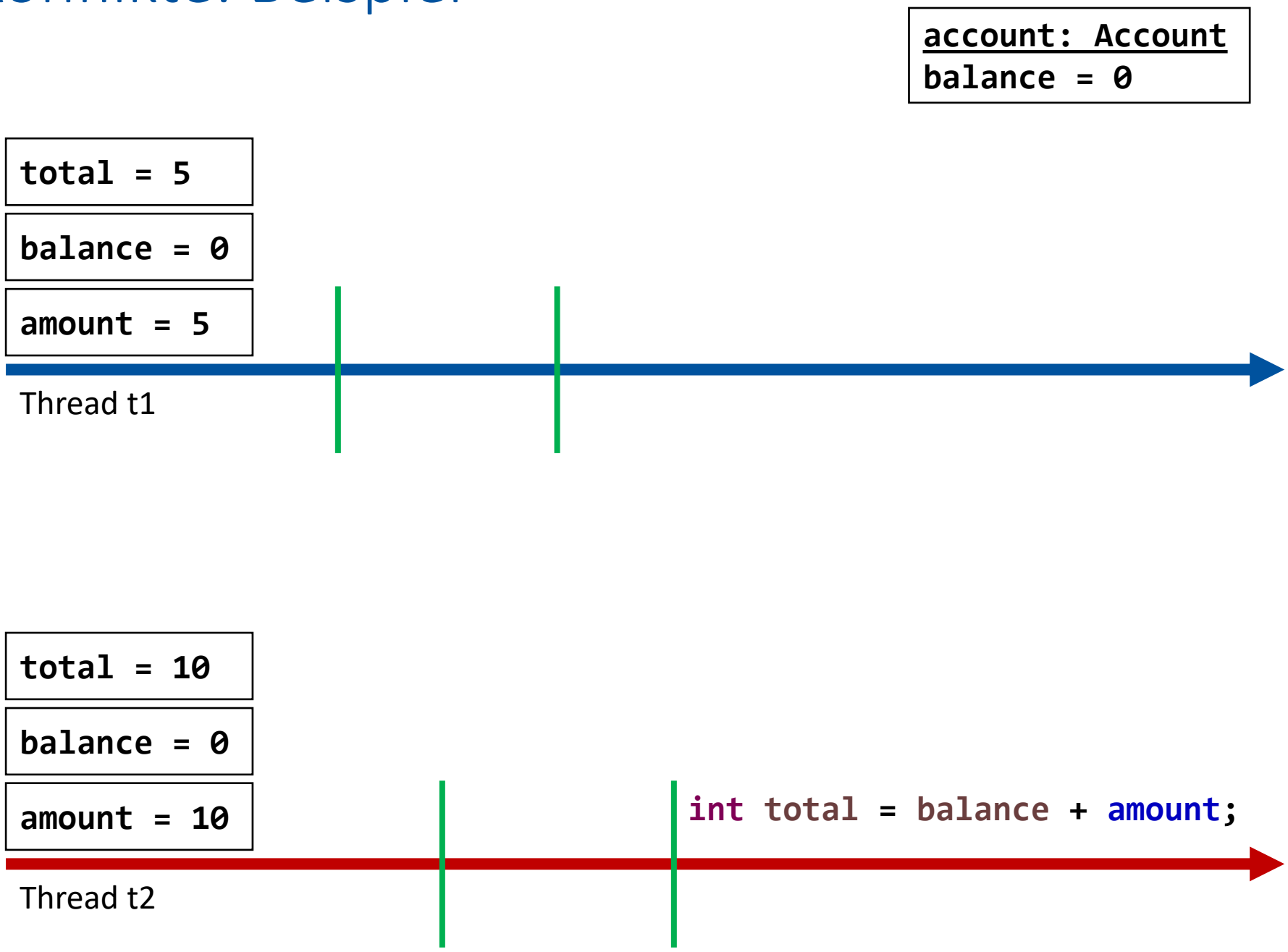
Konflikte: Beispiel



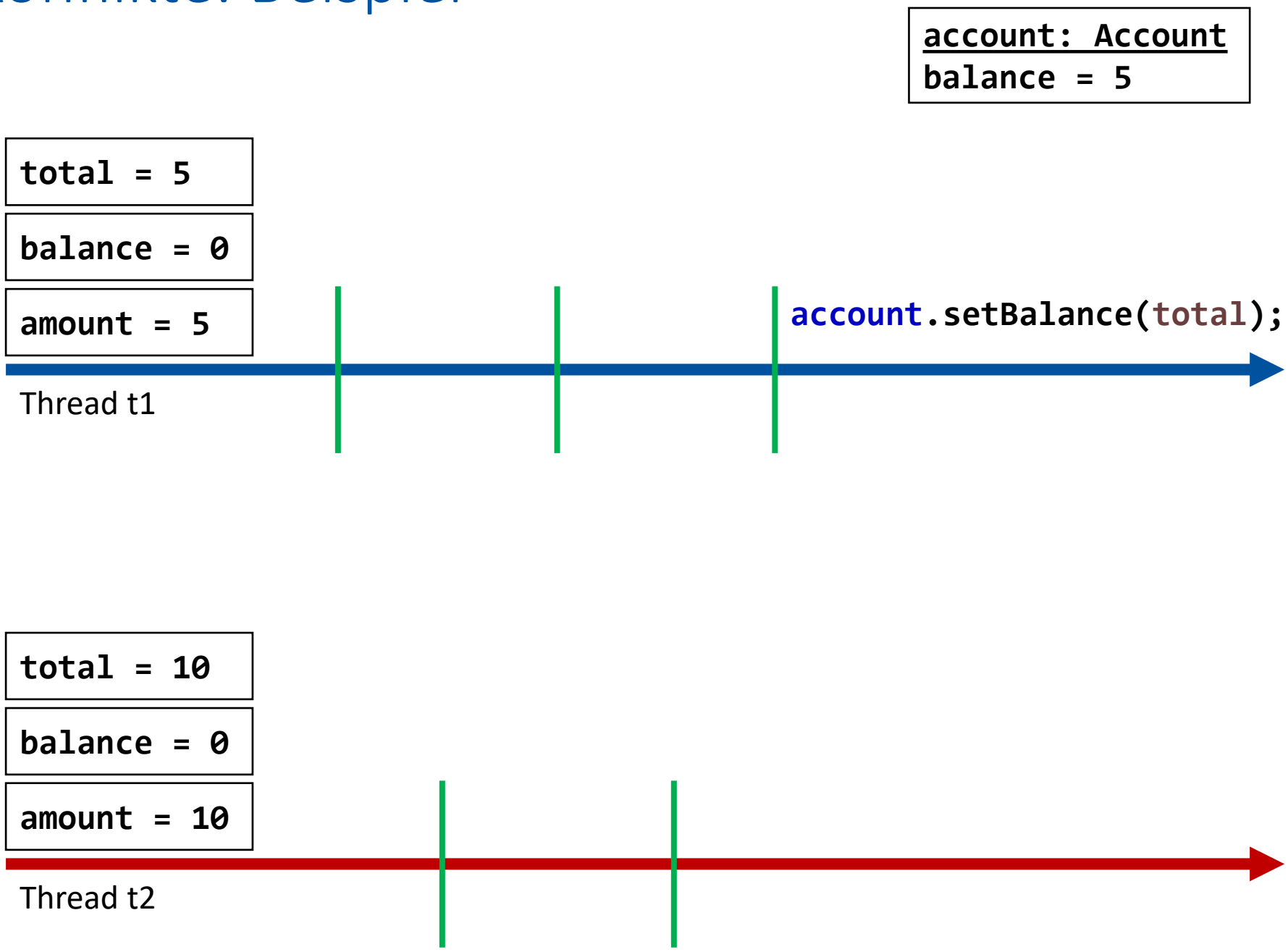
Konflikte: Beispiel



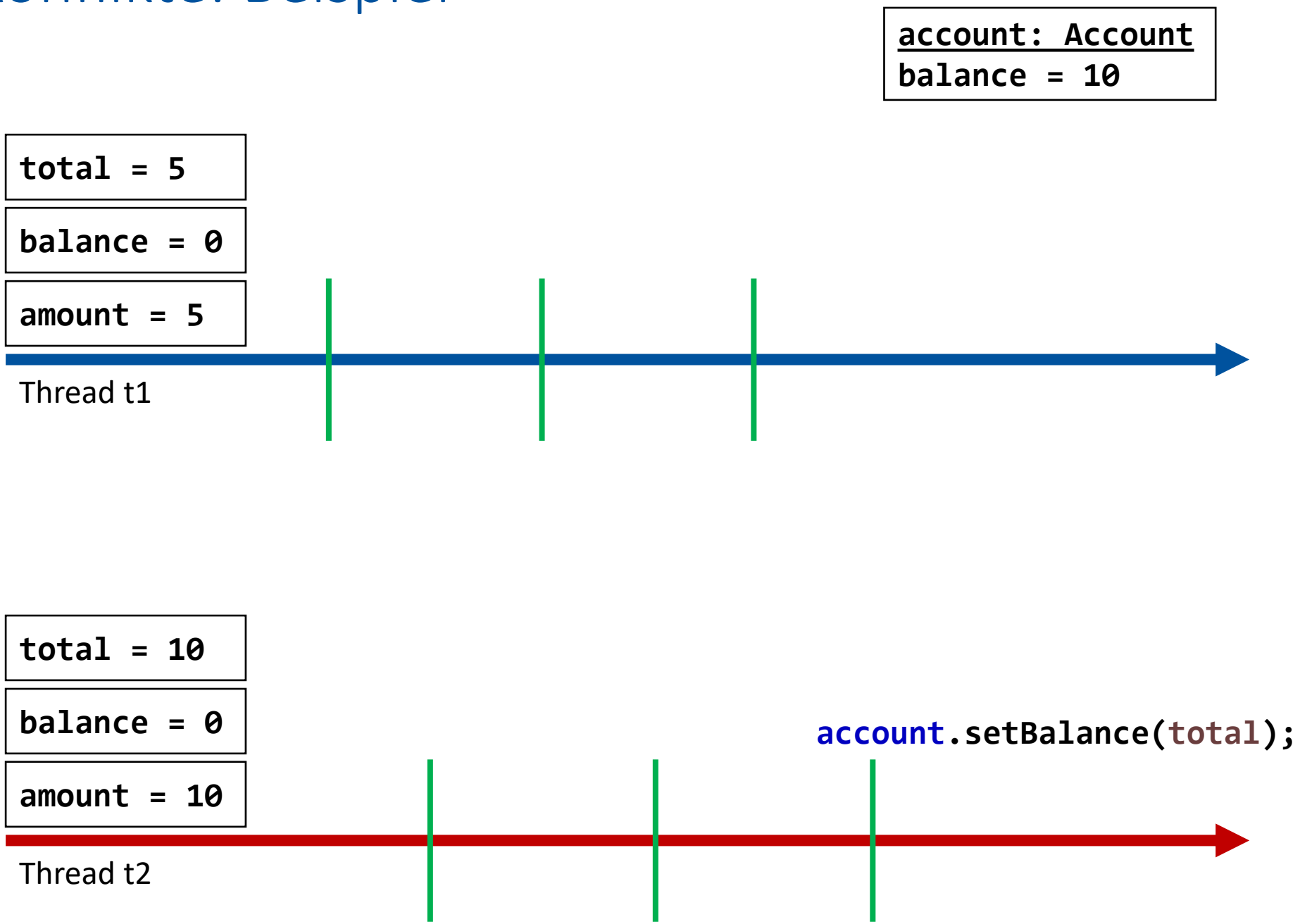
Konflikte: Beispiel



Konflikte: Beispiel



Konflikte: Beispiel



CARL VON OSSIETZKY universität OLDENBURG

Konflikte

- Konflikte beim Zugriff auf gemeinsame Ressourcen
  - Zugriff auf Variablen
  - Zugriff auf Dateien
  - Zugriff auf Netzwerk
  - Zugriff auf Hardware (Drucker o.ä.)
- Gegenseitiges Überschreiben von Werten
  - Variablen, Dateien
- Unkontrollierte Reihenfolge von Daten oder Anweisungen
  - Netzwerk, Hardware

→ Synchronisation der Zugriffe erforderlich

  - kein gleichzeitiger Zugriff
  - exklusiver Zugriff für einen Thread

Konflikte: Herausforderung

- Zwischen zwei beliebigen Anweisungen eines Threads kann potenziell ein anderer Thread beliebige Anweisungen ausführen
- Zwischen Auslesen und Schreiben einer Variablen kann eine Threadumschaltung erfolgen

```
x = 5;
x = x + 2;
```

- Zeile 2:
  - **x** wird eingelesen (Wert: 5)
  - **x** wird anderweitig verändert (Wert: 3)
  - **x** wird geschrieben (Wert: $5 + 2 = 7$)
  - anderweitige Veränderung (3) geht verloren

Kritische Abschnitte

- Prozesse/Threads markieren kritische Abschnitte (*critical sections*), in denen jeweils nur ein Prozess gleichzeitig aktiv sein kann
- Zugriff auf die geschützten Ressourcen darf nur innerhalb dieser kritischen Abschnitte erfolgen
- Ist ein Prozess A in einem kritischen Abschnitt, dann müssen alle anderen Prozesse mit dem Zugriff warten, bis A den Abschnitt verlassen hat
  - ➢ Erst wenn der Prozess A den kritischen Abschnitt verlässt, darf der nächste Prozess dort eintreten

CARL VON OSSIETZKY universität OLDENBURG

Kritische Abschnitte: **`synchronized`**

CARL VON OSSIETZKY universität OLDENBURG

synchronized

- Synchronisiert die Ausführung eines Code-Abschnittes oder einer ganzen Methode
- Synchronisation bezieht sich immer auf ein konkretes Objekt
- Beispiel: Liste mit synchronisierten **`add()`**, **`get()`** und **`remove()`** Methoden
  - keine gleichzeitige Ausführung der drei Methoden möglich, nur nacheinander
  - gleichzeitige Ausführung der Methoden von zwei unterschiedlichen Listen möglich

CARL VON OSSIETZKY universität OLDENBURG

`synchronized`: Beispiel

```
public class SynchronizedList<T> {

    private List<T> list = new ArrayList<>();

    public synchronized void add(T elem) {
        list.add(elem);
    }

    public synchronized T get(int index) {
        return list.get(index);
    }

    public synchronized void remove(T elem) {
        list.remove(elem);
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

`synchronized` (2)

- Synchronisierte Abschnitte sind immer bezüglich eines konkreten Objekts synchronisiert
  - Standardfall: Das Objekt, auf dem der Code (z.B. die Methode) ausgeführt wird
  - Auch möglich: Angabe des Objekts, über das synchronisiert werden soll

CARL VON OSSIETZKY universität OLDENBURG

`synchronized`: Beispiel (2)

```
public class SynchronizedList<T> {
    private List<T> list = new ArrayList<>();

    public void add(T elem) {
        synchronized(this) {
            list.add(elem);
        }
    }

    public T get(int index) {
        synchronized(this) {
            return list.get(index);
        }
    }

    public void remove(T elem) {
        synchronized(this) {
            list.remove(elem);
        }
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

synchronized: Beispiel (3)

```
public class SynchronizedHashList<T> {

    private List<T> list = new ArrayList<>();
    private List<Integer> hashList = new ArrayList<>();

    public void add(T elem) {
        synchronized(list) {
            list.add(elem);
        }
        synchronized(hashList) {
            hashList.add(elem.hashCode());
        }
    }

…
```

CARL VON OSSIETZKY universität OLDENBURG

`synchronized` Methoden

- **`synchronized`** gehört nicht zur Signatur
  - ➢ muss beim Überschreiben oder Überladen nicht übernommen werden

- Der Code

```
public synchronized int size() {
    return list.size();
}
```

entspricht

```
public int size() {
    synchronized(this) {
        return list.size();
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

`static synchronized` Methoden

- Methoden, die statisch und synchronisiert sind, werden nicht über eine Instanz synchronisiert, sondern über die Klasse
- Dadurch können diese Methoden instanzunabhängig nur ein einziges Mal gleichzeitig ausgeführt werden

Verschachtelte Synchronisation

- Synchronisationsblöcke können ineinander verschachtelt werden, auch wenn sie auf dem gleichen Objekt synchronisieren

```
Object lock = new Object();
synchronized(lock) {
    synchronized(lock) {
        synchronized(lock) {
            System.out.println("This works.");
        }
    }
}
```

- Java löst die doppelten Synchronisationen automatisch auf
- Das funktioniert auch über Methodenaufrufe hinweg, wenn innerhalb der aufgerufenen Methode wieder auf dem gleichen Objekt synchronisiert wird

CARL VON OSSIETZKY universität OLDENBURG

Synchronisationsregeln

- Synchronisation ist notwendig, wenn mehrere Threads auf gemeinsame Daten zugreifen und mindestens ein Thread die Daten verändert
- Wann synchronisieren?
  - einer schreibt, andere lesen
- Was synchronisieren?
  - alle lesenden und schreibenden
- Wo synchronisieren?
  - an der Stelle, an der gelesen oder geschrieben wird

CARL VON OSSIETZKY universität OLDENBURG

Verwendung von Synchronisation

- Synchronisation ist aufwendig
  - Planung
  - Programmierung
- Synchronisation ist teuer
  - ineffizient bei der Ausführung
  - verringert Geschwindigkeitsvorteile der Parallelisierung

➢ Synchronisation nur dann, wenn sie nötig ist

- nicht vorsichtshalber alles synchronisieren

➢ Synchronisation so feingranular wie möglich

- kritische Abschnitte möglichst klein halten

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Parallelität und Nebenläufigkeit
- Threads in Java
- Kommunikation zwischen Threads
- Synchronisation