Programmieren 3

Kapitel 7 Threads

- Motivation
- Die Klasse Thread
- Konkurrierende Zugriffe auf Objekte und Variablen
- Synchronisation
- Kommunikation zwischen Threads
- Erzeuger Verbraucher Problem
- Thread-Verwaltung

Was ist eigentlich ein Thread?

Weitere Details zu Threads in **Betriebssysteme** und **Verteilte Verarbeitung**

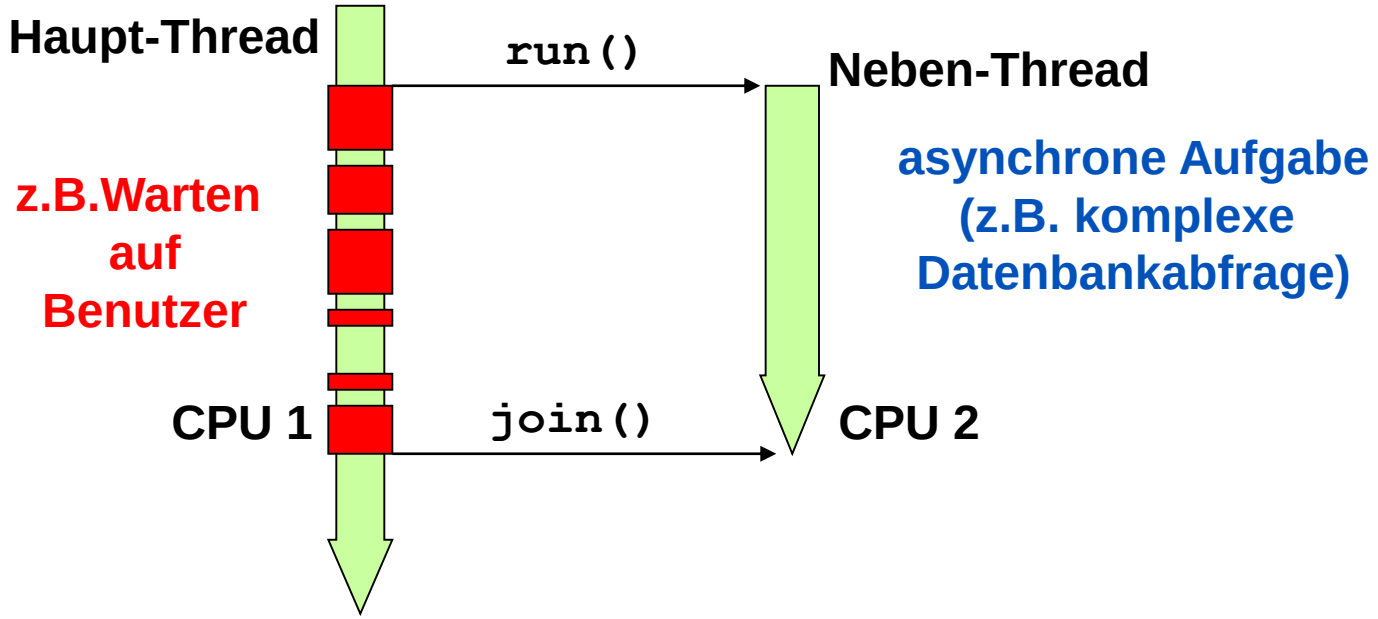
- Java verwendet Threads (Stränge).
  - Ein Thread (thread of control) ist ein sequentieller Ablauf innerhalb eines Programms.
  - Parallelität entsteht, wenn mehrere Threads gleichzeitig aktiv sind.
  - Beispielsweise kann ein Thread ein Bild laden, während ein anderer bereits beginnt, es am Schirm anzuzeigen.
- Von Threads zu unterscheiden sind Prozesse.
  - Prozesse verfügen jeweils über einen eigenen Speicherbereich im Hauptspeicher und über eigene Statusinformation wie Registerbelegung oder geöffnete Dateien.
  - Alle zu einem Programm gehörigen Threads benutzen denselben Speicherbereich. Man nennt Threads daher auch leichtgewichtige Prozesse.
  - Da Threads denselben Speicherbereich benutzen, können sie sich gegenseitig stören.

Warum eigentlich Threads verwenden?

- Bessere Interaktion mit dem Benutzer
- Simulation von simultanen Aktivitäten / Implementierung paralleler Algorithmen
- Ausnutzung von Mehrprozessor-Maschinen
- Verfügbarkeit des Systems bei langsamen Operationen
  (durch asynchrone Aufrufe)
- Steuerbarkeit: Abläufe starten, unterbrechen, fortsetzen, abbrechen
- Werden von einigen Programmiersprachen sehr gut unterstützt

Ausnutzen von Parallelität

- Die Java Virtual Machine startet in einem eigenen Prozess
  - Ein erster `Thread` startet und endet mit der Ausführung von `main()`
  - Ein zweiter `Thread` kümmert sich um die Garbage Collection
  - Der Programmierer erzeugt beliebig weitere `Thread`s
- Jede Anwendungen hat einen Haupt-Thread, der weitere erzeugen kann.



Haben Threads auch Nachteile?

- Nutzen muss gegen die Kosten abgewogen werden:
  - Ressourcenverbrauch (Speicher, etc.)
  - Komplexität (→ Entwicklungsaufwand, Fehlerwahrscheinlichkeit)
  - Effizienz (Konstruktion von Threads, Scheduling, Context Switch, etc.)
- Threads müssen aufeinander abgestimmt werden (sonst Gefahr von inkonsistenten Zuständen und Deadlocks)
- Nichtdeterminismus: Durch Scheduling können Threads beliebig verzahnt ausgeführt werden.
- Der wechselseitige Ausschluss (**mutual exclusion**) von Zugriffen auf eine gemeinsam benutzte Variable muss durch Freigeben der Variablen für einen Thread und Sperren für die übrigen implementiert werden (synchronized, wait, notify)

Die Klasse Thread

- Threads sind normale Java-Objekte. Die Klasse Thread liefert die nötigen Methoden.
- Die wichtigsten Konstruktoren

```
public Thread();
public Thread(String name);
public Thread(Runnable r)
```

- Über das Interface *Runnable* wird definiert, was ein Thread tut. Es besitzt nur eine Methode

```
public void run();    // interface Runnable
```

- Für benannten Threads kann der Name abgelesen und geändert werden mit

```
public final String getName();
public final void setName(String name);
```

- Durch ein neues Thread-Objekts wird ein noch inaktiver Thread erzeugt. Er wird dann gestartet mit

```
public void start();
```

Implementieren von Threads

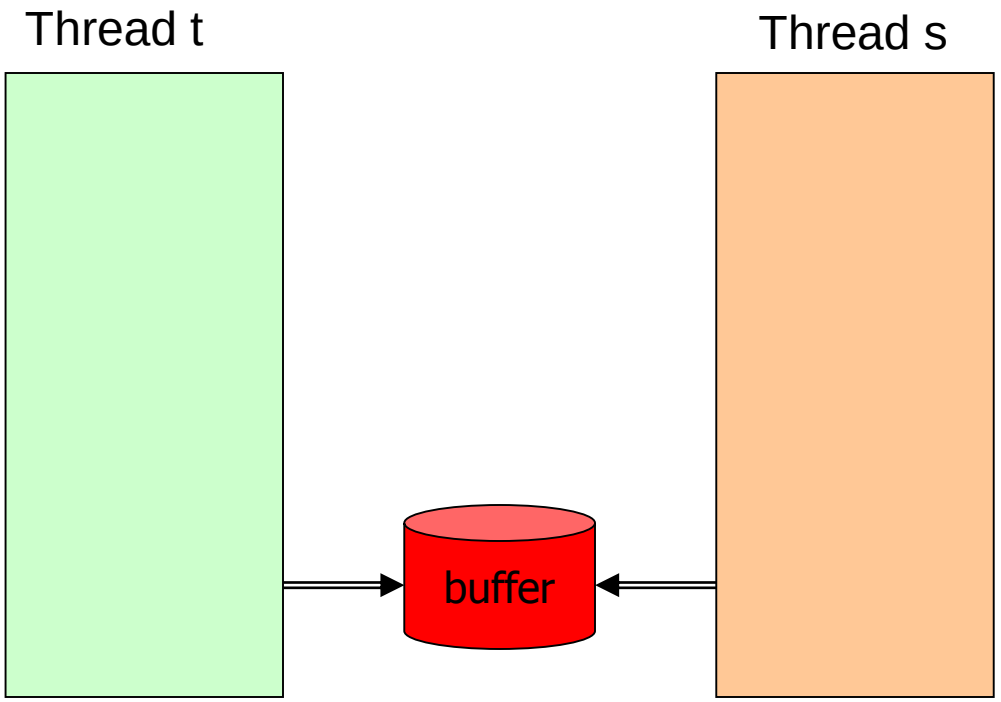
- Threads können auf 2 Arten implementiert werden:
  - Eigene Klasse schreiben und von Thread ableiten
  - Eigene Klasse schreiben, die das Interface **Runnable** implementiert:

```
interface Runnable {
     public void run ();
}
```

Beispiel `TwoThread`

```
public class TwoThread extends Thread {
	public void run() {
		for ( int i = 0; i < 10; i++ ) {
			System.out.println("New thread");
		}
	}

	public static void main(String[] args) {
		TwoThread tt = new TwoThread();
		tt.start();

		for ( int i = 0; i < 10; i++ ) {
			System.out.println("Main thread");
		}
	}
}
```

Zwei Threads, eine Ressource (1)



Problem: t und s kommen sich bei Buffer-Zugriffen in die Quere

Zwei Threads, eine Ressource (2)

- Unsere Ressource: ein Integer-Wert gekapselt in einer Klasse.
- Aufgabe: zwei Threads führen parallel die folgenden Schritte aus:
  - aktuellen Wert aus Ressource lesen
  - Delta addieren
  - neuen Wert zurückschreiben

```
class IntBuffer {
    private int value = 0;

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

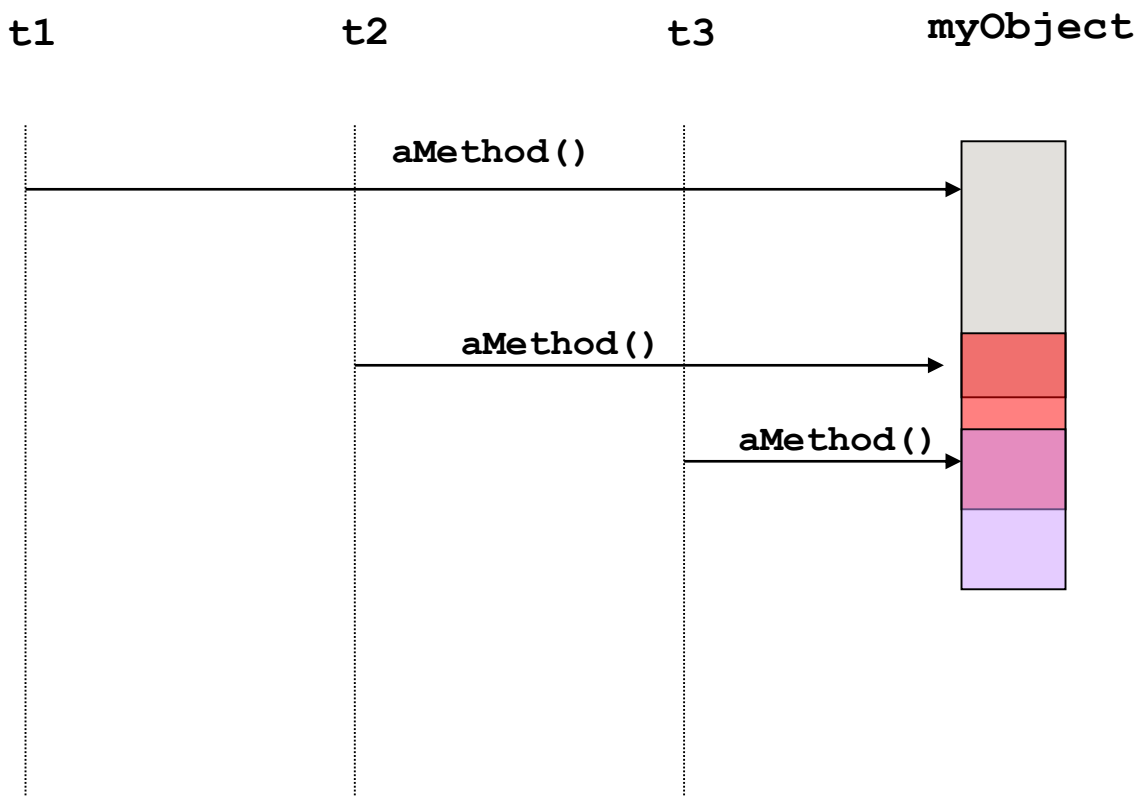
Zwei Threads, eine Ressource (3)

```
class AsyncMath extends Thread {
   ... // Attribute und Konstruktor
   public void addDelta() {
     try {
         int a = buffer.getValue();
         a += delta;
         sleep(waitPattern++ & 15);
         buffer.setValue(a);
     } catch (InterruptedException e) { }
   }
   public void run() {
       for (int i = 0; i < 1000; i++) {
          addDelta();
          System.out.println(buffer.getValue());
       }
   }

}
```

```
IntBuffer buffer = new IntBuffer();
Thread t1 = new AsyncMath(buffer, 1);
Thread t2 = new AsyncMath(buffer, -1);
t1.start(); t2.start();
try {
   t1.join(); t2.join();
} catch (Exception e) { }
System.out.print("Erg.:"+ buffer.getValue());
```

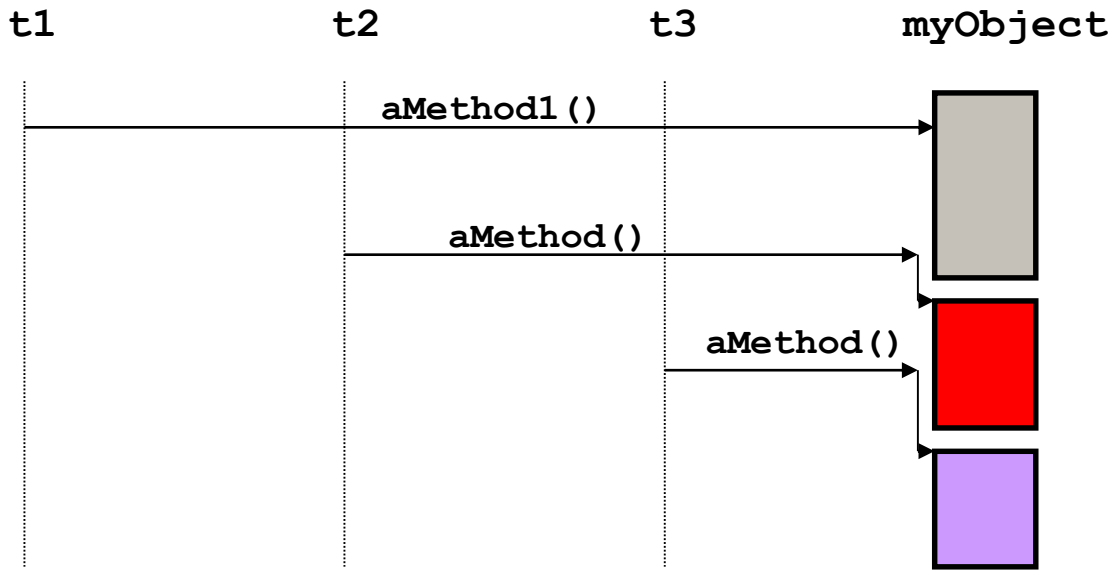
Synchronisation (1)

- Verschiedene Threads bearbeiten ein Objekt durcheinander: Kann katastrophale Folgen haben!



Synchronisation (2)

- Verschiedene Threads bearbeiten ein Objekt sequentiell
- Das Schlüsselwort `synchronized` ermöglicht das wechselseitige Ausschließen von Codekörpern
  - `synchronized` als Methoden-Modifizierer
  - `synchronized` um einem Anweisungsblock
  - `synchronized` als Modifier von Klassen-Methoden



Synchronisation über Block

```
public class MyClass {
      private Object myObject;

      public void myMethod() {
            synchronized(myObject) {

            // benutze hier myObject

            }
      }
}
```

- **Ablauf bei Synchronisation**
  - 1. Sperre das angesprochene Objekt
  - 2. Führe den synchronisierten Block durch
  - 3. Gebe das angesprochene Objekt wieder frei

Synchronisation über Methode

```
public class MyClass implements {

      public synchronized void myMethod( ... ) {
      ....
      }
      ...
}
```

- **Ablauf bei Synchronisation**
  - 1. Sperre das angesprochene Objekt
  - 2. Führe die synchronisierte Methode durch
  - 3. Gebe das angesprochene Objekt wieder frei
- **Wichtig**
  - Die Sperre betrifft nur synchronisierte Methoden.
  - Der sperrende Thread hat weiterhin Zugriff auf die gesperrten Methoden.

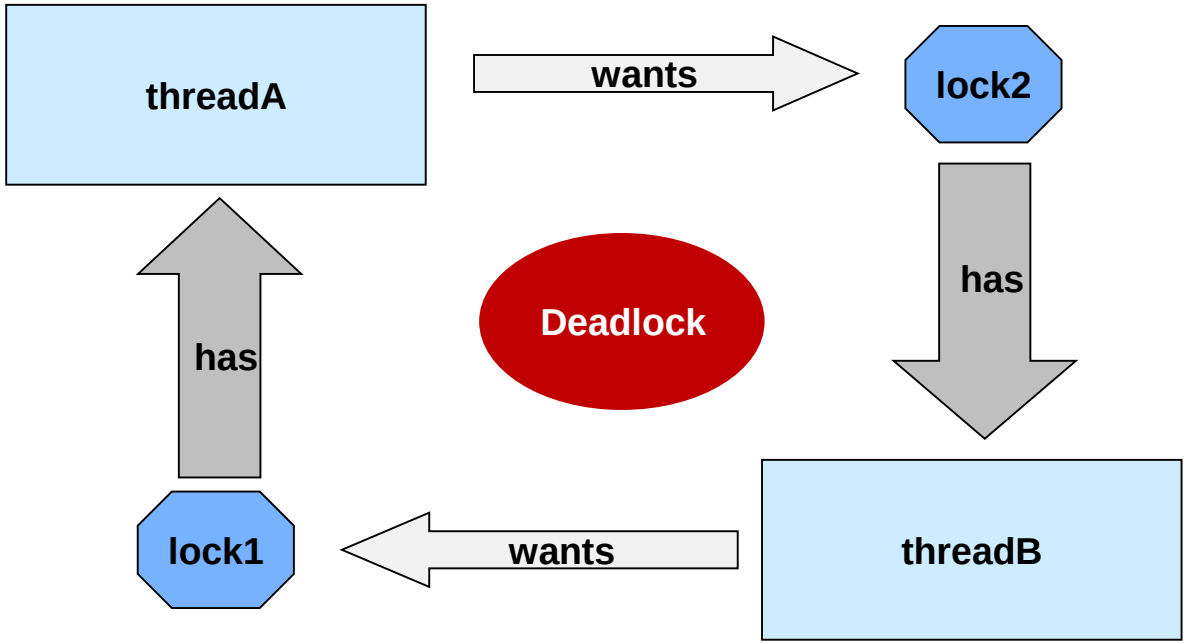
Zwei Threads, eine synchronisierte Ressource

```
class AsyncMath extends Thread {
   ... // Attribute und Konstruktor

   public void addDelta() {
       try {
         synchronized ( buffer ) {
            int a = buffer.getValue();
            a += delta;
            sleep(waitPattern++ & 15);
            buffer.setValue(a);
         }
       } catch (InterruptedException e) { }
   }
   ...

}
```

- Synchronisation über `synchronized (Object)`
- Im anschließenden Block hat immer nur genau ein Thread Zugriff auf das `Object`.
- Alle anderen müssen warten!

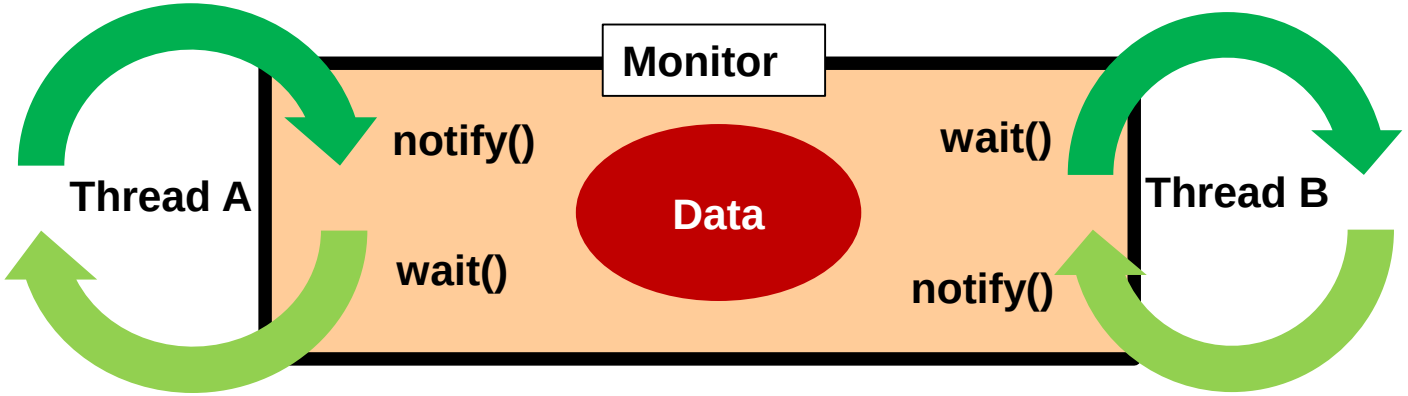
Wann `synchronized` nicht gut genug ist

- Durch Synchronisation kann ein `Thread` sicher Werte ändern, die von einem anderen `Thread` gelesen werden können
- Was noch fehlt ist ein Mechanismus, der es einem `Thread` erlaubt solange zu warten, bis er eine Meldung von einem anderen `Thread` erhält



Kommunikation zwischen `Threads`

- Der *wait/notify* Mechanismus ist schon in `Object` definiert
  - Das API besteht aus den Methoden `notify()`, `notifyAll()`, `wait()`, `wait(long)` und `wait(long, int)`
  - Alle Methoden sind als `final` deklariert



wait, notify, notifyAll

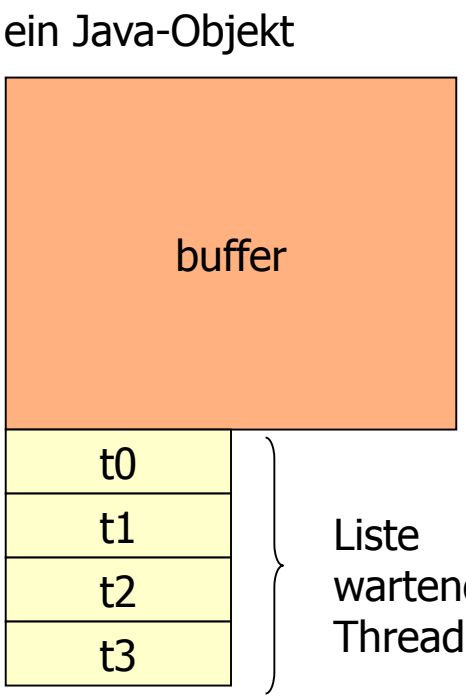
1. Java verwaltet für jedes Objekt eine Liste wartender THREADS

2. Ein Thread kommt dort rein durch den Aufruf von wait() während das Objekt synchronisiert ist:

```
synchronized(buffer) {
    ....
    buffer.wait();
}
```

3. **notify()** weckt den ersten Thread,
   notifyAll() weckt alle Threads dieser Liste

4. wait, notify und notifyAll sind Methoden von Object, stehen also immer zur Verfügung. Sie funktionieren allerdings nur, wenn man exklusiven Zugriff auf das Objekt hat
   Beachte: sleep, join sind Methoden von Thread!



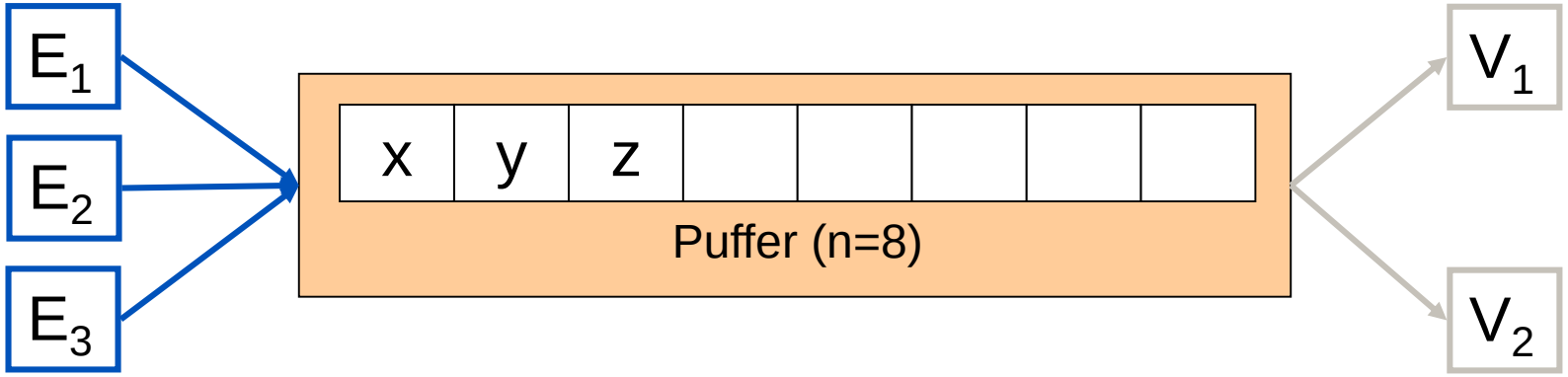

```
public class X {
 …
   public synchronized doY() {
      wait();
   }
 …
}
```

oder

```
synchronized(obj) {
   obj.wait();
}
```

Erzeuger/Verbraucher-Problem

- Jeder Erzeuger/Verbraucher greift auf denselben Puffer zu
- Jeder Erzeuger/Verbraucher läuft in einem eigenen `Thread`
- Jeder Erzeuger kann ein Objekt im Puffer ablegen, wenn Platz ist
- Jeder Verbraucher kann ein Objekt nur entnehmen, wenn der Puffer nicht leer ist
- Erzeuger/Verbraucher warten, wenn ihre Bedingungen nicht erfüllt sind



Das Einfügen von Objekten in den Puffer

```
public class Buffer {
    private int items = 3;            // Items im Puffer
    private int spaceleft = 5;        // Platz im Puffer

    public synchronized void put(Object x) {
        while (spaceleft == 0) {
            try {
                wait();
             } catch(InterruptedException e) {}
        }

        // jetzt ist Platz, um obj in den Puffer einzutragen ...

        items++;
        spaceleft--;
        notifyAll();
    }
}
```

Das Auslesen von Objekten in den Puffer

```
public class Buffer {
    // ...
    public synchronized Object get() {
    while (items == 0) {
            try {
                wait();
             } catch(InterruptedException e) {}
        }

        // jetzt kann ein Objekt aus dem Puffer entnommen werden

        items--;
        spaceleft++;
        notifyAll();

        return obj;
    }
}
```

Der Erzeuger

```
public class Producer implements Runnable {
    private int id;
    private Buffer b;

    public Producer (Buffer b, int id) {
        this.b  = b;
        this.id = id;
    }
    public void run() {
        while(true) {
            // Erzeuge ein Produkt x
            b.put(x);
        }
    }
    public void startProducer() {
        Thread t = new Thread(this);
        t.start();
    }
}
```

Der Verbraucher

```
public class Consumer implements Runnable {
    private int id;
    private Buffer b;

    public Consumer(Buffer b, int id) {
        this.b = b;
        this.id = id;
    }
    public void run()  {
        while(true){
            x = b.get();
            // mach was mit dem geholten Produkt x
        }
    }
    public void startConsumer() {
        Thread t = new Thread(this);
        t.start();
     }
}
```

Ein Hauptprogramm

```
public class ConsumerProducer {
    static private final int M = 4, N = 3;

    public static void main (String args[]) {
        Buffer b = new Buffer ();

        for (int i = 0; i < M; i++)  {
            Consumer c = new Consumer (b, i);
            c.startConsumer();
        }
        for (i = 0; i < N; i++) {
            Producer p = new Producer (b, i);
            p.startProduer();
        }
    }
}
```

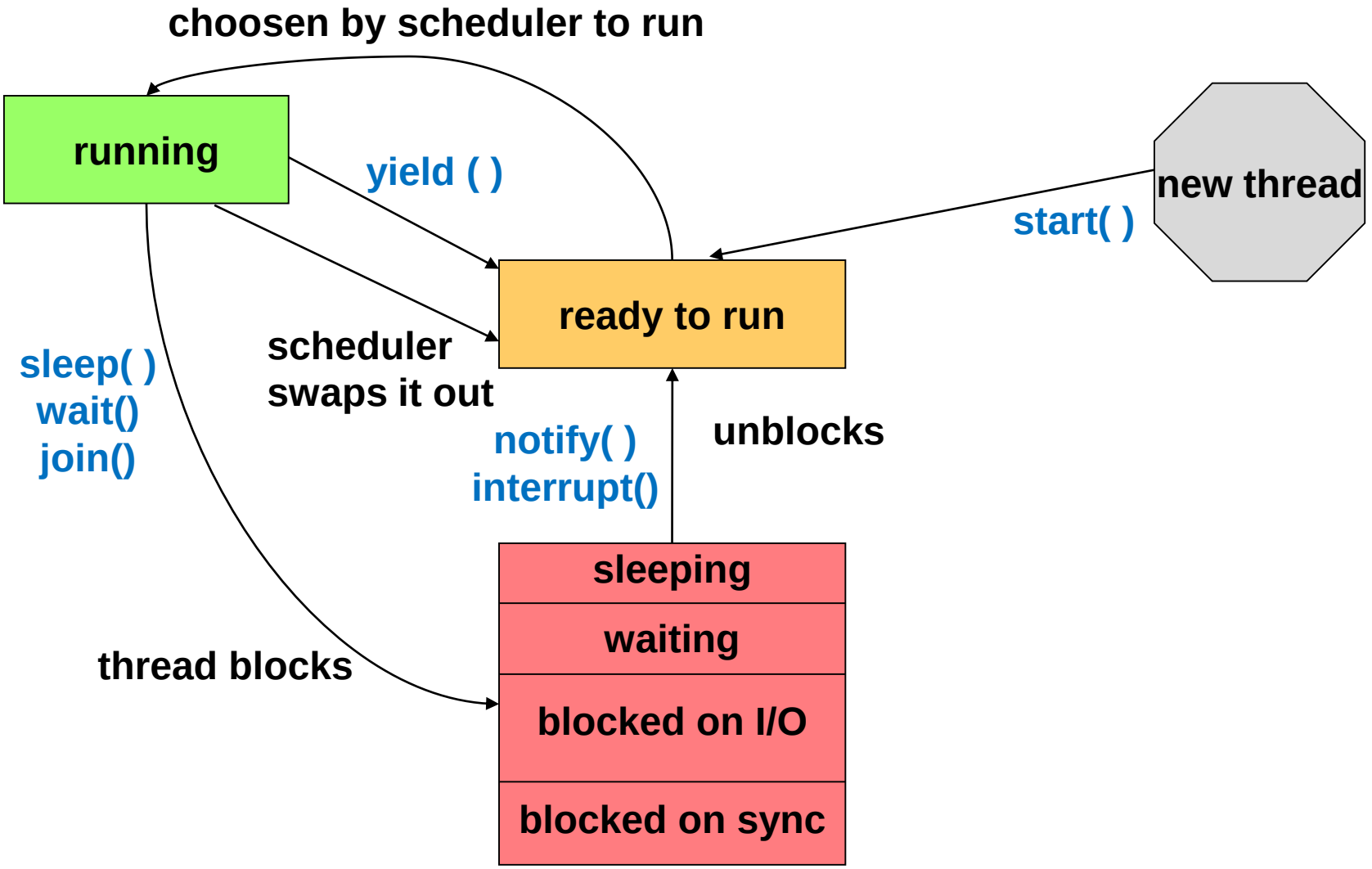
Thread-Verwaltung (1)

- Der Thread endet mit der Beendigung der `run`-Methode
- Ob ein Thread bereits gestartet und noch nicht beendet wurde, prüft man mit
  - `public final boolean isAlive();`
- Den momentan aktiven Thread erhält man mit der Klassenmethode
  - `public static Thread currentThread();`

Thread-Verwaltung (2)

- `sleep(milli), sleep(milli, nano)` : Der rufende Thread wartet maximal die angegeben Zahl von Milli/Nanosekunden.
- `join(), join(milli), join(milli, nano)` : Der rufende Thread wartet, bis der angesprochene Thread endet, aber maximal die angegeben Zahl von Milli/Nanosekunden.
- `sleep` und `join` werfen die `InterruptedException`. Diese wird durch die Methode `interrupt` ausgelöst.
- `yield` gibt potentiell die Kontrolle an einen anderen Thread ab.

Übergangsdiagramm von `Thread`-Zuständen



Thread-Beispiel (`sleep` und `interrupt`)

```
class SweetDreams extends Thread {

  public void run() {

    try {
        sleep(10000000);
        System.out.println("sanft aufgewacht :-)");
    } catch (InterruptedException e) {
        System.out.println("unsanft aufgewacht :-(");
    }
   }

}
```

```
...
SweetDreams dreams = new SweetDreams();
dreams.start();
sleep(N?);
dreams.interrupt();
...
```

Das Stoppen von Threads

- Das taktvolle Stoppen
  - Unterbrechen eines Threads: `interrupt()`, `isInterrupted()`
  - Unterbrechung des aktuellen Threads: `Thread.interrupted()`
  - Ausnahme bei Unterbrechung: `InterruptedException`
  - Freiwillige Aufgabe der Verarbeitung: `Thread.yield()`

- Das abrupte Stoppen
  - Die *deprecated* Methoden `suspend()`, `resume()` und `stop()`
    → nicht mehr verwenden!

Thread-Beispiel (`join`)

```
class WaitAndPrint extends Thread {

  private char c;
   private Thread otherThread;

  public WaitAndPrint(char c, Thread otherThread) {
       this.c = c;
       this.otherThread = otherThread;
   }

  public void run() {
       if (null != otherThread)
       try {
           otherThread.join();
       } catch (InterruptedException e) { }

     for (int i = 0; i < 1000; i++)
         System.out.print(c);
   }

}
```

```
WaitAndPrint px = new WaitAndPrint('-', null);
  WaitAndPrint py = new WaitAndPrint('|', px);
  px.start();
  py.start();
```

```
WaitAndPrint px = new WaitAndPrint('-', null);
  WaitAndPrint py = new WaitAndPrint('|', null);
  px.start();
  py.start();
```

Bugpattern für Threads: Double Checked Locking

- **Problem:** in einer Multithread-Umgebung soll ein Singleton instanziiert werden.

```
public static Singleton getInstance() {
   if (instance == null) {
         instance = new Singleton();
   }
   return instance;
 }
```

Versuch 1

```
public static synchronized Singleton getInstance() {
   if (instance == null) {
         instance = new Singleton();
   }
   return instance;
 }
```

Versuch 2

Versuch 3

```
public static Singleton getInstance() {
   if (instance == null) {
      synchronized(Singleton.class) {
         instance = new Singleton();
      }
   }
   return instance;
 }
```

Versuch 4

```
public static Singleton getInstance() {
   if (instance == null) {
      synchronized(Singleton.class) {
         if (instance == null)
            instance = new Singleton();
      }
   }
   return instance;
 }
```

Funktioniert das???

Zusammenfassung Threads

- Threads sind ein wichtiges und häufig verwendetes Konzept in der fortgeschrittenen Programmierung
- Threads müssen sauber programmiert werden sonst kann es zu schwer auffindbaren Programmfehlern führen
- In Java werden Threads durch die Sprache sehr gut unterstützt
- Zum wechselseitigen Ausschluss beim konkurrierenden Zugriff auf Objekte gibt es Sprachkonstrukte (synchronized, wait, notify)
- Das Erzeuger/Verbraucher-Problem kann durch Threads gelöst werden