Threads

• Bisher rein **sequentielle** Programme
• **Parallele** Programme erlauben die Durchführung mehrerer gleichzeitiger Aktionen
• **Threads** sind parallele Prozesse innerhalb eines Hauptprogrammes
• Die Ausführung erfolgt mit einem Prozessor nur quasiparallel
•Code muss **reentrant** sein, d. h. gleichzeitig in mehreren Tasks verwendet werden können

Threads

- Zur Erstellung ist ein Elternprozess („main-Thread“) erforderlich

```
public class ThreadBsp1 {
    public static void main (String args[]) {
    //Den Namen lesen
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    //Den Namen verändern
    t.setName("My Thread");
    System.out.println("After name change: " + t);
    }
}
```

Ablaufkontrolle von Threads

- `init()` Thread erzeugen
- `start()` Starten des Thread
- `sleep()` Schlafzustand

- Ein Thread muss selbst warten oder sich selbst beenden !! Daher kein Resume oder Stop aus einem anderen Thread heraus. Laufender Thread kann mit interrupt() bzw. isInterrupted() prüfen, ob eine Unterbrechungsanforderung vorliegt. Interrupt() löscht hierbei das Anforderungsflag.

Threads – Zustandswechsel

- `wait()`

stellt Thread wartend bis ein anderer Thread notify() aufruft

- `notify()`

informiert wartende Threads, dass eine Veränderung stattgefunden hat (ein wartender Thread läuft weiter, aber keine FIFO-Verwaltung !)

- `notifyAll()`

informiert alle wartenden Threads. Thread mit höchster Priorität läuft weiter – auf gleicher Prioritätsebene eigene Mechanismen zur Auswahl einführen

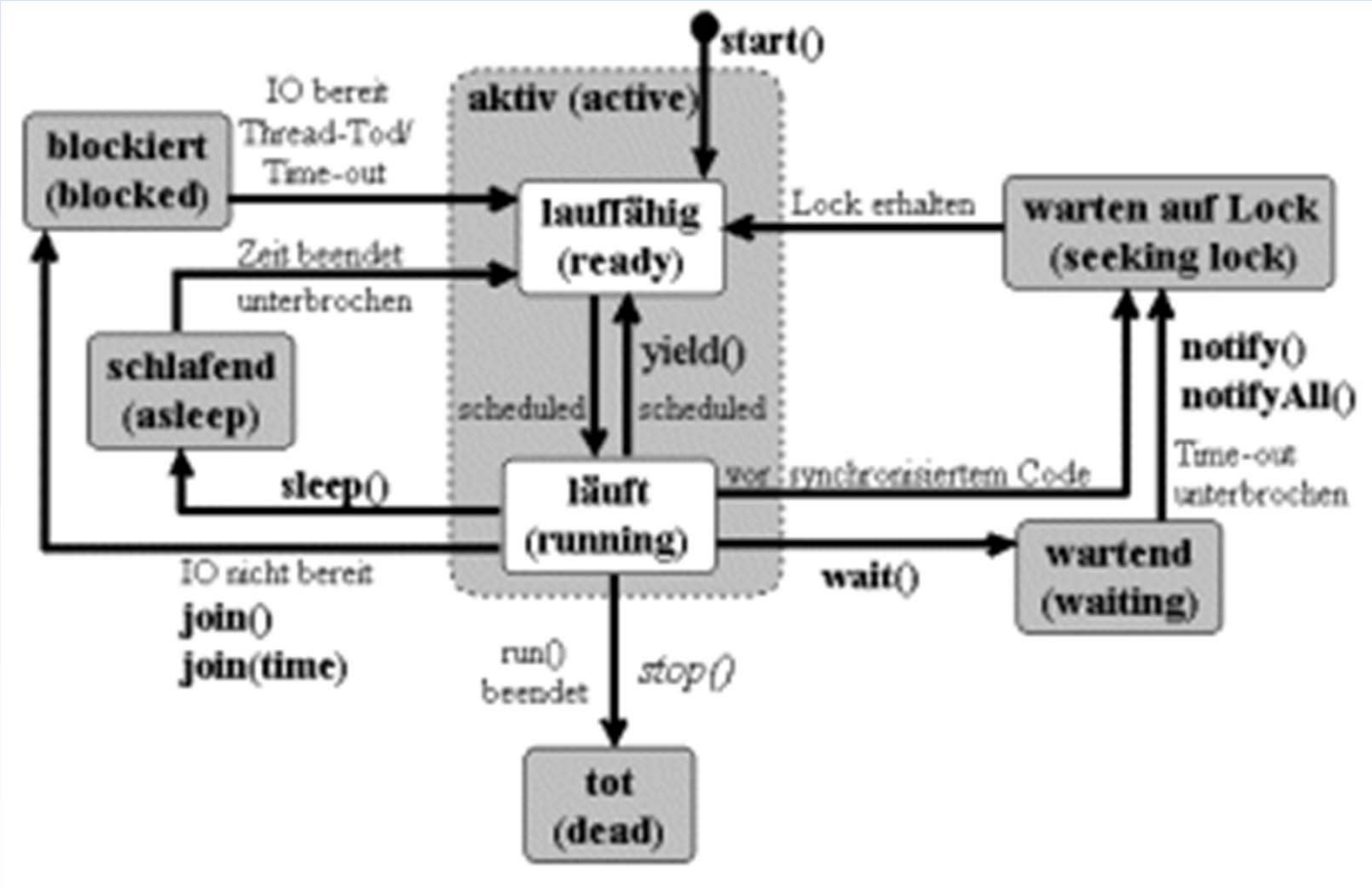
- `yield()`

Thread gibt CPU ab, ohne wartend gestellt zu werde

- `join()`

wartet auf Beendigung eines Threads

Thread-Zustände



Threads – Vererbung mit extends

1. Klassendefinition mit Erweitern der Thread-Klasse

```
public class MyThread extends Thread{
public void run(){
        }
}
```

2. Implementierung der Methode run() mit Programmcode, der quasi-parallel ausgeführt wird. In „run“ kann auch eine Endlosschleife realisiert sein.

3. Erzeugung eines Threads t1 mittels

```
Thread t1= new MyThread();
//Überprüfen, ob Erzeugung erfolgreich
    if (t1 != null) {t1.start}
                  else {Fehlerbehandlung}
```

4. Starten des Thread mit der Methode start(), normalerweise in main

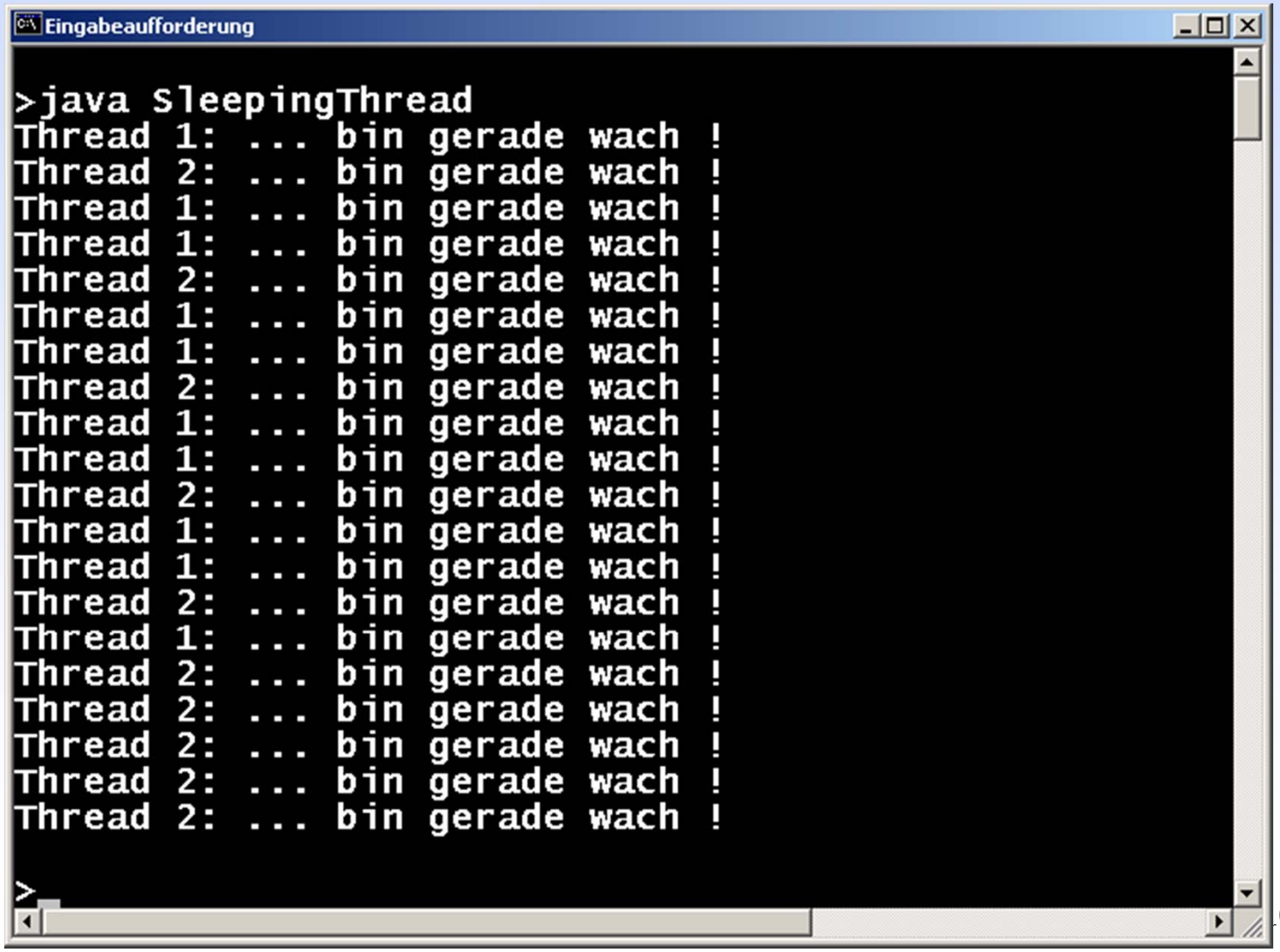
Threads - Beispiel

```
class myThread extends Thread {
        String name;
        int pause;
        public myThread(String text, int dauer)
{
                name=text;
                pause=dauer;
        }
        public void run(){
            for (int i=0;i<1000;i++) {
             System.out.println(name +
              ": ... bin gerade wach!");
              try {sleep(pause);
              } catch (InterruptedException e)
                 {}
              }
      }
}
```

Threads - Beispiel

```
public class ThreadBsp{
public static void main (String[] args) {
      Thread t1=new myThread("Thread 1",50);
      Thread t2=new myThread("Thread 2",100);
      t1.start();
      t2.start();
      }
    }
```

Output: Multi-Threading



Threads – Vererbung mittels interface

1. Klassendefinition mit Erweitern der Thread-Klasse

```
public class MyThread implements Runnable{
public void run(){
        }
}
```

2. Erzeugen eines Threads t1 mittels

```
1. MyThread t1interface= new MyThread();
2. Thread t1=new Thread(t1interface);
3. //Überprüfen, ob Erzeugung erfolgreich
4.     if (t1 != null) {t1.start}
5.                 else {Fehlerbehandlung}
```

3. Starten des Thread mit der Methode start()

Threads – Interface

```
public class MainThread {

  public static void main(String[] args) {
    //Instanzen in 2 Stufen anlegen
    myThread ThreadNr1 = new myThread(1);
    Thread t1= new Thread(ThreadNr1);
    t1.start();

    myThread ThreadNr2 = new myThread(2);
    Thread t2= new Thread(ThreadNr2);
    t2.start();
  }
}
```

Threads - Interface

```
class myThread implements Runnable{
  String name;
  int nr=0;
  int k;
  myThread() {
    nr=0;
  }
   myThread(int i) {
    nr=i;  //Threads können durchnummeriert werden
  }

  public void run(){
    if (nr==0)
      System.out.println("Neuer Thread gestartet");
      else
        {
          System.out.println("Neuer Thread mit Nummer "+nr+„
                gestartet");
        }
        for (int i=0;i<1000;i++) {
        System.out.println(nr + ": ... bin gerade wach!");
try {Thread.sleep(10);} catch (InterruptedException e) {}
     }
  }
}
```

Thread – Synchronisation

- **Problemstellung: Zugriff auf gemeinsame Variablen durch mehrere Threads**

- **Eigenschaft „synchronized“ verhindert einen Threadwechsel – „unteilbare Aktion“**

- **Ab Java 5 ist die Alternative „java.util.concurrent.locks.Lock“**

Threads – Synchronisation

```
public class Main {

public static void main (String[] args) {
              Datenbereich datenInMain = new
Datenbereich();

              datenInMain.datensetzen(0,0);

      Thread t1=new myThread(1,50,datenInMain);
      Thread t2=new myThread(2,100,datenInMain);
                    t1.start();
                    t2.start();

      }
 }
```

Threads – Synchronisation

```
class myThread extends Thread {
        int nr;
        int pause;
         Datenbereich datenImThread;

public myThread(int nr, int dauer, Datenbereich daten)
        {
                this.nr=nr;
                pause=dauer;
                DatenImThread=daten;
        }
           public void run(){
                for (int i=1;i<100;i++) {
                    datenImThread.datensetzen(nr,Pause);
                    System.out.println(nr + " : ... bin
gerade wach! Datenwerte: " + datenImThread.x +" " +
datenImThread.y);

                          try {sleep(pause);
                          } catch (InterruptedException e) {}
                }
        }
 }
```

Threads – Synchronisation

Datenbereich nicht geschützt

```
public class Datenbereich {
    int x,y;

    public void datensetzen(int a, int b){
        x=a;
         for (int i=1;i<1000;i++){}
        y=b;
    }
    public Datenbereich() {
    }
}
```

Threads – Synchronisation

Datenbereich geschützt – Methode sperren

```
public class Datenbereich {
    int x,y;

   public synchronized void datensetzen(int a, int b){
        x=a;
         y=b;
…
…
…

    }
    public Datenbereich() {
    }
}
```

Threads – Synchronisation

Datenbereich geschützt – instanzenabhängig

```
public class Datenbereich {
    int x,y;

    public void datensetzen(int a, int b){
        synchronized(this)
        {
            x=a;
            y=b;
        }
       …
       …
       …

    }
    public Datenbereich() {
    }
}
```

Threads – Synchronisation

Mittels Sperrobjekt – auch kaskadierbar

```
Beispiel für die Synchronisation mit einem Sperr-
Instanz:

class SteuerKlasse {}

class MeineKlasse {
 SteuerKlasse sperrInstanz = new SteuerKlasse() ;
 ....
 public void Methode() {
  synchronized(sperrInstanz) {
// kritischer Bereich
//Programmcode der synchronisiert werden soll
 }
}
```

Thread – Beenden

- **Kein Beenden durch anderen Thread – führt zu Inkonsistenzen**
- **Sondern: Abfrage einer Stop-Bedingung innerhalb des Threads**

Threads – Beenden (1)

```
class myThread extends Thread {
private boolean running=true;
int i=0;
public void run(){
  while(isRunning()) {
  i++;
  System.out.println("Hello World (“ + i +")");
  }
}
public synchronized void stopRunning()
        {
            running = false;
        }
public synchronized boolean isRunning()
        {
            return running;
        }
}
```

Threads – Beenden (2)

```
public class StopThread {

   public static void main(String[] args)
       {
              myThread1 t = new myThread1();
              t.start();
              try
              {
                     Thread.sleep(5000);
              }
              catch(InterruptedException e)
              {
              }
              if (t.isRunning()) {
                     t.stopRunning();
              }
       }
}
```

Thread – Synchronisation

- **Fortführen eines Threads in** Abhängigkeit **von** Bedingungen**, die ein anderem Thread gesetzt werden**
- **Verwendung von** Semaphoren

Threads – Synchronisation

```
synchronized void doWhenCondition()
{
while ( ! condition )
try
      {
      wait();
      }
      catch(InterruptedException e) {}
// Wartet bis notify aufgerufen wird
}
```

Threads – Synchronisation

```
synchronized void changeCondition()
{
…
// Veränderung passiert
notify();
}
```

Threads – Semaphore

•**Semaphore -** Warteschlange in Verbindung mit Zähler
•Ist die Resource nicht frei, wartet der entsprechende Thread

•Zugriff **mittels P() –** bei Nichterfolg wartend stellen
•Freigabe **mittels V()** – evtl. Wartende freigeben

Es wird unterschieden zwischen
• Mutual Exclusion (MUTEX – gegenseitiger Ausschluss)
• Binäre Semaphore (Zähler +1, -1)
• Mehrwertige Semaphore (Zähler +n, -n, als unteilbares Ereignis)
• Semaphorgruppen (mehrere Semaphore werden geschaltet, als unteilbares Ereignis z. B. „2 Gabeln“ beim Philosophenproblem)

Threads – Synchronisation

Deadlock

• Gegenseitige Abhängigkeiten von Threads

• Programm läuft nicht mehr weiter
• Vermeidung durch genaue Modellierung der gegenseitigen Abhängigkeiten

Threads – Synchronisation

Thread Scheduling

• Threads laufen nur quasiparallel

•Austausch der Threads geschieht durch Scheduling-Algorithmus

• Scheduling bei Java ist prioritätsorientiert

•Mittels `setPriority()` kann die Priorität zwischen MIN_PRIORITY und MAX_PRIORITY eingestellt werden

Threads – Semaphore

- **Beispiele**
- **Aufgaben**

Threads – Atomare Operationen

Ab Java 5

• Auswahl an atomaren Operationen werden zur Verfügung gestellt

•Einbinden von java.util.concurrent.atomic.AtomicInteger

Threads – Atomare Klassen

Ab Java 5

• Auswahl an atomaren Klassen und zugehöriger Methoden
•Einbinden von java.util.concurrent.atomic.AtomicInteger
•Z. B.
•AtomicBoolean
•AtomicInteger
•AtomicLong
•AtomicIntegerArray
•Etc.

Threads – Atomare Methoden

Ab Java 5

- `get()`
- `set(value)`
- `getAndSet()`
- `getAndAdd(delta)`
- `getAndDecrement()`
- etc.

Beispiel

```
…
public void run(){
        …
                b.set(5);
        …
                b.getAndIncrement();
}
```