Einführung und Grundbegriffe

- Sequentielles System
  - Rechner führt einen einzigen Strom von Anweisungen aus
- Paralleles System
  - Anweisungen können gleichzeitig ausgeführt werden
  - Beispiel: Mehrprozessor-Systeme, Rechnernetz
- Nebenläufiges System (*concurrent system*)
  - Es existieren mehrere Ströme von Anweisungen, die unabhängig voneinander abgearbeitet werden
  - Diese Ströme werden entweder parallel oder pseudoparallel ausgeführt
  - Pseudoparallelität: Durch häufigen Wechsel zwischen den Vorgängen wird Gleichzeitigkeit vorgetäuscht
- Verteiltes System
  - räumliche (oder auch konzeptionelle Aufteilung) der einzelnen Komponenten eines Systems

Typische nebenläufige Systeme

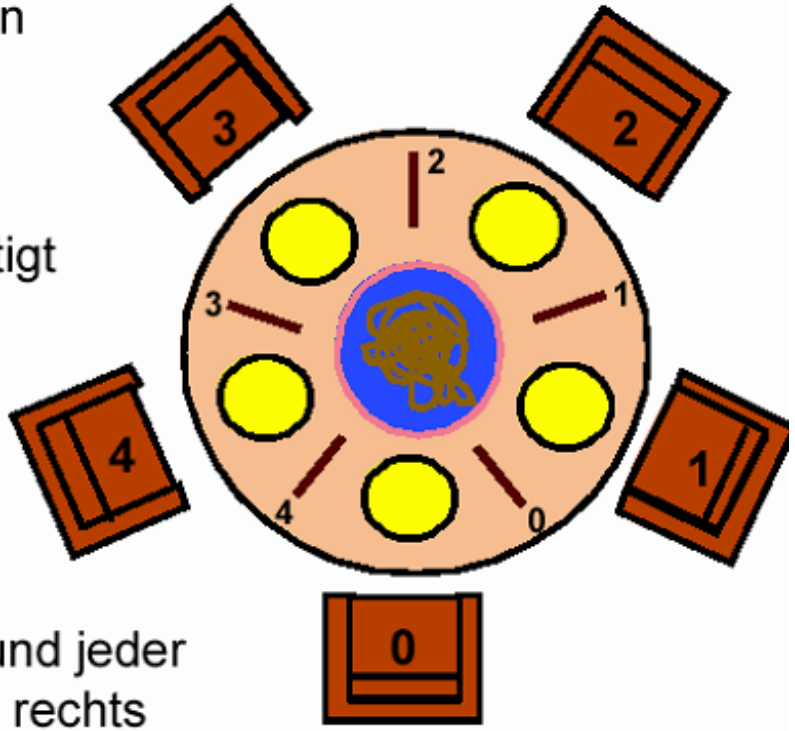
- Betriebssysteme
  - Prozesse verschiedener Nutzer laufen nebenläufig ab
- Technische Systeme
  - Steuerung und Regelung technischer Systeme, meist im Echtzeitbetrieb
  - permanentes Reagieren auf ankommende Nachrichten
- Systeme auf Parallelrechnern
  - Bsp.: rechenzeitintensive Simulationen
- Web-Services
  - permanente Annahme und Ausführung von Diensten auf dem Server
  - Bsp.: Fahrplanauskunft
- Graphische Oberflächen
  - Permanente Reaktion auf ankommende Events

Prozess und Thread

- **Prozess**
  - Vorgang mit eigenen Ressourcen (z.B. Adressraum)
  - wird vom Betriebssystem verwaltet
- **Thread**
  - Vorgang, der innerhalb eines Prozesses abläuft und die Ressourcen dieses Prozesses nutzt
  - wird vom Benutzerprogramm (z.B. Java Virtual Machine) verwaltet
  - *lightweigt process*
- Wenn wir an der Modellierung von Vorgängen interessiert sind, unterscheiden wir meist nicht zwischen Prozess und Thread

Beispiel: The Dining Philosophers *

Fünf Philosophen sitzen um einen runden Tisch. Jeder Philosoph beschäftigt sich entweder mit Denken oder Essen. In der Mitte des Tisches steht eine große Schüssel Spaghetti. Ein Philosoph benötigt zwei Gabeln, um essen zu können.



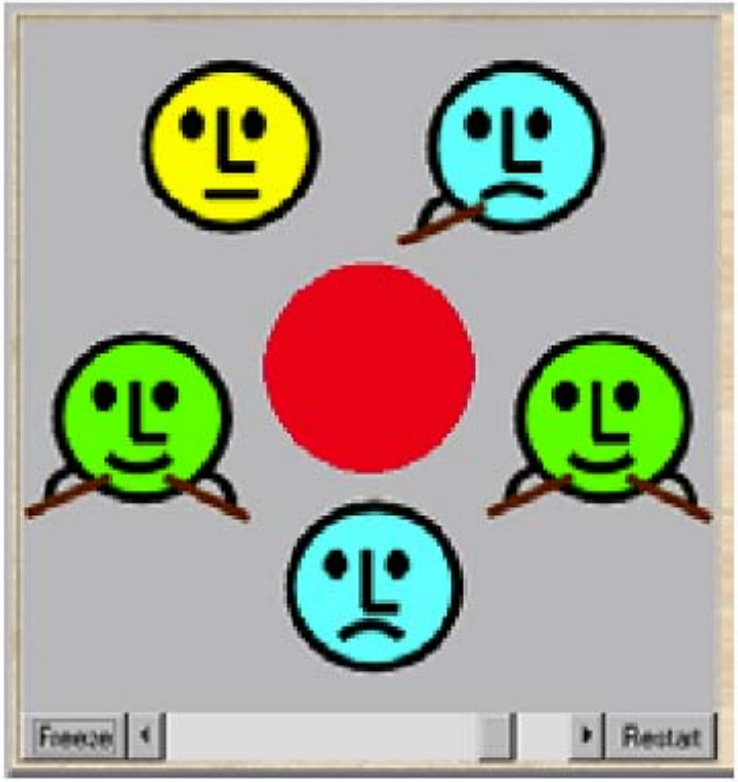
Zwischen jedem Teller liegt eine Gabel und jeder Philosoph kann nur die Gabeln links und rechts von seinem Teller nehmen.

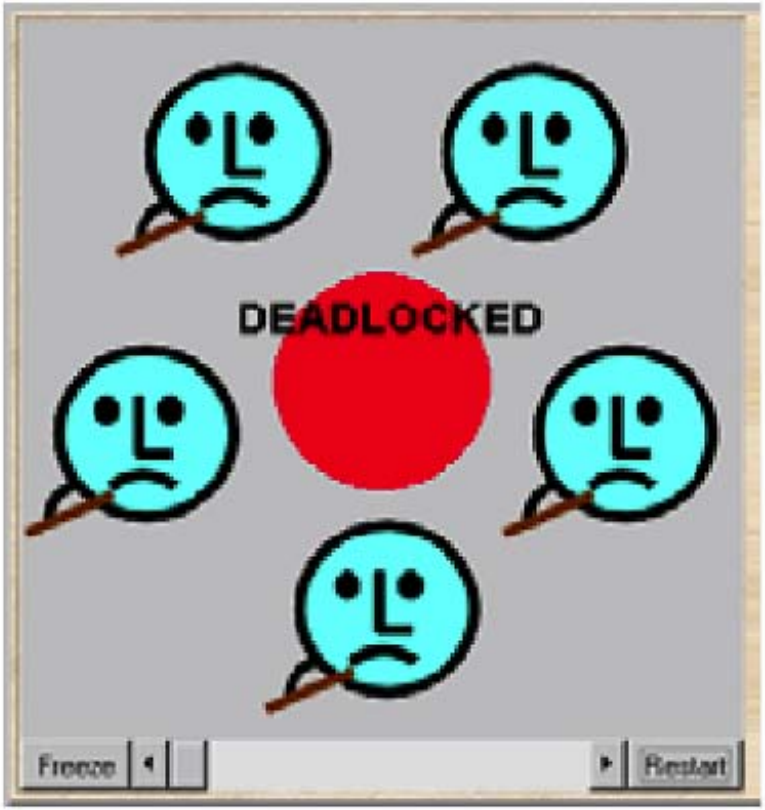
* Dijkstra, 1968

Was kann passieren?

- Ein Philosoph, der hungrig ist, muss so lange warten, bis die Gabeln links und rechts von seinem Teller frei sind.
- Es kann passieren, dass ein Philosoph nie zum Essen kommt, auch wenn er hungrig ist.
  - dieser Philosoph wird (im wahrsten Sinn des Wortes) **ausgehungert**
  - die Verteilung der Gabeln ist nicht **fair**
- Es kann auch passieren, dass alle Philosophen gleichzeitig hungrig sind und alle die Gabel an ihrer linken Seite nehmen
  - dann kann kein Philosoph jemals essen
  - das System befindet sich in einer **Verklemmung** (*deadlock*)

Beispielsituationen





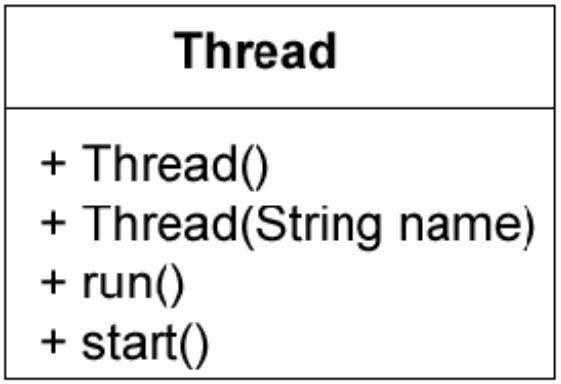
Java Threads

- Beim Start eines Java-Programms wird ein Prozess erzeugt, der u.a. einen Thread enthält, der die **main**-Methode der angegebenen Klasse ausführt.
- Programmcode weiterer, vom Anwendungsprogramm definierter Threads: in **run**-Methoden

```
public void run () {
  // Code, der in eigenem Thread ausgeführt wird
}
```

- 2 Möglichkeiten für die Definition der **run**-Methode
  - Ableiten der Klasse **Thread**
  - Implementieren der Schnittstelle **Runnable**

Ableiten der Klasse Thread

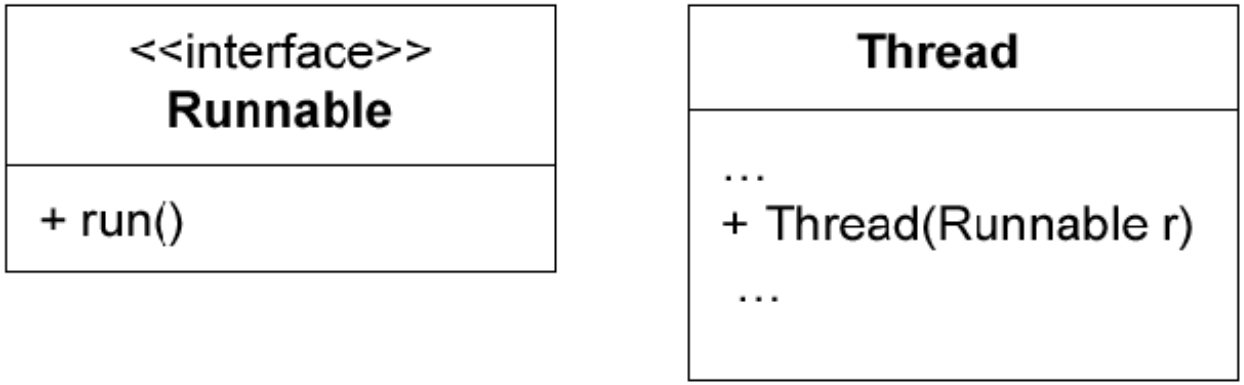


in package java.lang

```
public class MyThread extends Thread  {

      public void run()  {
      System.out.println("Hello World");
      }

      public static void main(String[ ] args)  {
          MyThread t = new MyThread();
          t.start();
      }
}
```

Start eines neuen Threads, der die run-Methode ausführt

Implementieren der Schnittstelle Runnable




```
public class SomethingToRun implements Runnable {
    public void run()
    {
        System.out.println("Hello World");
    }

    public static void main(String[] args)
    {
        SomethingToRun runner = new SomethingToRun();
        Thread t = new Thread(runner);
        t.start();
    }
}
```

Beispiel: Klasse Loop1

```
public class Loop1 extends Thread {

   public Loop1(String name) {
      super(name);
   }

   public void run() {
      for(int i = 1; i <= 100; i++)
      {
            System.out.println(getName() + " (" + i + ")");
      }
   }

   public static void main(String[] args) {
      Loop1 t1 = new Loop1("Thread 1");
      Loop1 t2 = new Loop1("Thread 2");
      Loop1 t3 = new Loop1("Thread 3");
      t1.start();
      t2.start();
      t3.start();
   }
}
```

Starten dreier Threads

Mögliche Ausgabe von Loop1

```
Thread 1 (1)
Thread 1 (2)
Thread 1 (3)
Thread 1 (4)
...
Thread 1 (35)
Thread 2 (1)
Thread 3 (1)
Thread 1 (36)
Thread 2 (2)
Thread 3 (2)
Thread 2 (3)
Thread 3 (3)
Thread 2 (4)
Thread 3 (4)
Thread 2 (5)
Thread 3 (5)
Thread 1 (37)
Thread 2 (6)
Thread 1 (38)
...
```

- Die drei Threads laufen zeitlich verzahnt ab
  - die JVM schaltet zwischen den Threads um
- Keine Aussage über die relative Geschwindigkeit der Threads möglich
  - Wann zwischen den Threads umgeschaltet wird, hängt vom Betriebssystem, Hardware, Version des JDKs und anderen Faktoren ab
- Ausgabe des Programms ist **nichtdeterministisch**

Probleme beim Zugriff auf gemeinsam genutzte Objekte

- Mehrere Threads kooperieren durch Zugriff auf gemeinsam genutzte Objekte
- Beispiel:
  - Klasse Konto mit einem Attribut für den Kontostand
  - Klasse Bank, die ein Feld von Konten enthält
  - Bankangestellte, die Beträge auf/von Konten buchen, werden als Thread modelliert (Klasse Bankangestellter)
  - Die Klasse Bankbetrieb enthält eine Bank und mehrere Bankangestellte, die auf dieses Objekt zugreifen
- Der Zugriff auf gemeinsame Objekte wird über die Konstruktoren realisiert

Beispiel: Klasse Konto und Klasse Bank

```
class Konto  {
   private float kontostand;
   public void setzen(float betrag)  {
       kontostand = betrag;
   }
   public float abfragen()  {
       return kontostand;
   }
}

class Bank  {
   private Konto[ ] konten;
   public Bank()  {
       konten = new Konto[100];
       for(int i = 0; i < konten.length; i++)  {
             konten[i] = new Konto();
       }
   }
   public void buchen(int kontonr, float betrag) {
       float alterStand = konten[kontonr].abfragen();
       float neuerStand = alterStand + betrag;
       konten[kontonr].setzen(neuerStand);
   }
}
```

Beispiel (Forts.): Klasse Bankangestellter

```
class BankAngestellter extends Thread  {
   private Bank bank;

   public BankAngestellter(String name, Bank bank) {
        super(name);
        this.bank = bank;
        start();
   }

   public void run() {
        for(int i = 0; i < 10000; i++)
        {    /* Kontonummer einlesen; simuliert durch Wahl einer Zufallszahl
                zwischen 0 und 99  */
             int kontonr = (int)(Math.random()*100);

             /* Überweisungsbetrag einlesen; simuliert durch Wahl einer Zufallszahl
               zwischen -500 und +499  */
             float betrag = (int)(Math.random()*1000) - 500;

             bank.buchen(kontonr, betrag);
        }
   }
```

Threadobjekt selbst startet die Ausführung der run-Methode

Beispiel (Forts.): Klasse Bankbetrieb

```
public class Bankbetrieb
{
   public static void main(String[] args)
   {
       Bank sparkasse = new Bank();
       BankAngestellter müller =
           new BankAngestellte("Andreas Müller", sparkasse);
       BankAngestellter schmitt =
           new BankAngestellte("Petra Schmitt", sparkasse);
   }
}
```

Die Angestellten greifen auf das gleiche Bank-Objekt zu

Was kann passieren?

1. Der Thread „Andreas Müller“ will 100 Euro auf Konto Nr. 44 buchen
   - Aufruf der Methode bank.buchen(44, 100)
   - Innerhalb dieser Methode wird die Methode abfragen() aufgerufen, mit dem zurückgegebenen Wert 50
   - Die Variable neuerStand wird auf 150 gesetzt
2. Es wird auf den Thread „Petra Schmitt“ umgeschalten; Petra Schmitt will 20 Euro von Konto Nr. 44 abbuchen
   - Aufruf der Methode bank.buchen(44, -20)
   - Innerhalb dieser Methode wird die Methode abfragen() aufgerufen, mit dem zurückgegebenen Wert 50
   - Die Variable neuerStand wird auf 30 gesetzt
   - Auf dem Kontoobjekt wird setzen(30) aufgerufen
3. Es wird wieder auf den Thread „ Andreas Müller“ umgeschalten
   - Die Ausführung der Methode bank.buchen(44, 100) wird fortgesetzt: auf dem Kontoobjekt wird setzen(150) aufgerufen

Fazit

- In dem eben beschriebenen Szenario ist das Abbuchen von 20 (Euro) „verlorengegangen“
- Dieser Effekt tritt zwar nur selten ein, das Programm ist jedoch fehlerhaft
- Eine Reduzierung Methode buchen(int kontonr, float betrag) auf eine einzige Java-Anweisung löst das Problem nicht
  - Java-Anweisungen werden in Folgen von Anweisungen auf der JVM übersetzt

Weiterer Lösungsversuch

- Zu einem Zeitpunkt darf nur ein Bankangestellter die Methode **buchen** ausführen
- Benutzung eines Sperrattributs **gesperrt** in der Klasse **Bank**

Klasse Bank (neue Version)

```
class Bank  {
   private Konto[ ] konten;
   private boolean gesperrt;

   public Bank()  {
        konten = new Konto[100];
        for(int i = 0; i < konten.length; i++)  {
              konten[i] = new Konto();
        }
        gesperrt = false;
   }
   public void buchen(int kontonr, float betrag) {
        while(gesperrt);
        gesperrt = true;
        float alterStand = konten[kontonr].abfragen();
        float neuerStand = alterStand + betrag;
        konten[kontonr].setzen(neuerStand);
        gesperrt = false;
   }
}
```

Probleme

1. Die Parallelität wird unnötig eingeschränkt
   - paralleles Buchen auf unterschiedlichen Konten ist unkritisch
2. Schlechte Effizienz der Lösung
   - unnötig viel Rechenzeit wird auf den Test der Variable gesperrt verwendet (sog. „busy waiting“ oder „aktives Warten“)
3. Die Lösung ist falsch
   - Wenn die Variable **gesperrt** false ist, können mehrere Threads die Anweisung *while(gesperrt)* passieren, bis die Variable auf true gesetzt wird

synchronized-Methoden

```
class Bank  {
   private Konto[ ] konten;
   public Bank()  {
        konten = new Konto[100];
        for(int i = 0; i < konten.length; i++)  {
              konten[i] = new Konto();
        }
   }
   public synchronized void buchen(int kontonr, float betrag) {
        float alterStand = konten[kontonr].abfragen();
        float neuerStand = alterStand + betrag;
        konten[kontonr].setzen(neuerStand);
   }
}
```

Bedeutung von synchronized

- Jedes Objekt in Java besitzt eine **Sperre**
- Soll eine mit synchronized gekennzeichnete Methode ausgeführt werden,
  - muss die Sperre gesetzt werden, wenn sie noch frei ist
  - ist die Sperre bereits gesetzt, wird der aufrufende Thread blockiert
  - der blockierte Thread wird beim Umschalten nicht berücksichtigt (passives Warten)
- Nach Beendigung der Methode wird die Sperre wieder freigegeben
  - Ein Thread, der auf die Freigabe wartet, kann jetzt fortgesetzt werden
- Also kann jeweils nur ein Thread die synchronized-Methode ausführen

Zulassen von parallelen Buchungen

- 1. Variante:
  - Methode buchen in Klasse Bank ohne **synchronized**
  - synchronized-Methode buchen in Klasse Konto

```
class Konto  {
   private float kontostand;

   public synchronized void buchen (float betrag)  {
       kontostand+ = betrag;
   }
}

class Bank  {
   private Konto[ ] konten;
   …
   public void buchen(int kontonr, float betrag) {
       konten[kontonr].buchen(betrag);
   }
}
```

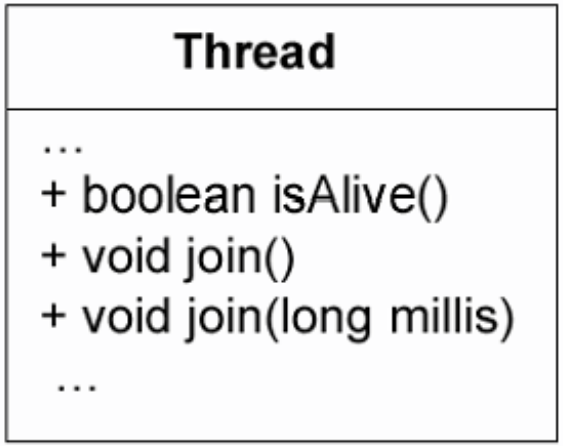
Wann muss eine Methode als synchronized gekennzeichnet werden?

- synchronize-Statements sind nicht umsonst
  - Sperrmechanismus kostet Zeit
  - synchronize-Statements können zu Verklemmungen führen
- Regel:

Wenn von mehreren Threads auf ein Objekt zugegriffen wird,
wobei mindestens ein Thread den Zustand des Objekts ändert,
dann müssen alle Methoden, die auf den Zustand des Objekts lesend
oder schreibend zugreifen, mit **synchronized** gekennzeichnet werden

Ende von Java Threads

- Ein Prozess (Programmlauf) ist zu Ende, wenn alle Threads zu Ende sind (incl. der Ausführung der main-Methode)



- isAlive testet, ob der aufgerufene Thread gestartet und noch nicht beendet wurde
- Ein Thread kehrt aus dem Aufruf der join-Methode zurück, falls der Thread, auf dessen Ende gewartet wird, zu Ende gelaufen ist (oder die angegebene Zeit vergangen ist).

Beispiel: Asynchrone Beauftragung mit Abfragen der Ergebnisse

- In einem sehr großen boolean-Feld soll die Anzahl der true-Werte gezählt werden
  - die Aufgabe wird auf mehrere Threads aufgeteilt, wobei jeder Thread einen gewissen Bereich des Feldes absucht.
- Nachdem alle Threads fertig sind (warten mit **join**-Methode), werden die Zählergebnisse der Threads addiert

Beispiel (1): Thread-Klasse Service

```
class Service implements Runnable {
   private boolean[ ] array;
   private int start;
   private int end;
   private int result;

   public Service(boolean[ ] array, int start, int end) {
      this.array = array;
      this.start = start;
      this.end = end;
   }

   public int getResult() {
      return result;
   }

   public void run() {
      for(int i = start; i <= end; i++) {
         if(array[i])  result++;
      }
   }
}
```

Beispiel (2): Klasse AsynchRequest

```
public class AsynchRequest  {
   private static final int ARRAY_SIZE = 100000;
   private static final int NUMBER_OF_SERVERS = 100;

   public static void main(String[ ] args)  {
      // Startzeit messen
      long startTime = System.currentTimeMillis();

      // Feld erzeugen mit zufälliger Mischung aus true und false (jeder 10. Wert ist true)
      boolean[] array = new boolean[ARRAY_SIZE];
      for(int i = 0; i < ARRAY_SIZE; i++)  {
         if(Math.random() < 0.1)
            array[i] = true;
         else
            array[i] = false;
      }

      // Feld für Services und Threads erzeugen
      Service[ ] service = new Service[NUMBER_OF_SERVERS];
      Thread[ ] serverThread = new Thread[NUMBER_OF_SERVERS];
```

Beispiel (3): Klasse AsynchRequest (Forts.)

```
// Threads erzeugen
int start = 0;
int end;
int howMany = ARRAY_SIZE / NUMBER_OF_SERVERS;

for(int i = 0; i < NUMBER_OF_SERVERS; i++) {
   end = start + howMany - 1;
   service[i] = new Service(array, start, end);
   serverThread[i] = new Thread(service[i]);
   serverThread[i].start();
   start = end + 1;
}

// Synchronisation mit Servern (auf Serverende warten)
try
{
   for(int i = 0; i < NUMBER_OF_SERVERS; i++)
      serverThread[i].join();
}
catch(InterruptedException e)  { }
```

Beispiel (4): Klasse AsynchRequest (Forts.)

```
    // Gesamtergebnis aus Teilergebnissen berechnen
    int result = 0;
    for(int i = 0; i < NUMBER_OF_SERVERS; i++)
    {
       result += service[i].getResult();
    }

    // Endzeit messen
    long endTime = System.currentTimeMillis();
    float time = (endTime-startTime) / 1000.0f;
    System.out.println("Rechenzeit: " + time);

    // Ergebnis ausgeben
    System.out.println("Ergebnis: " + result);
  }
}
```

Beispielergebnisse

NUMBER_OF_SERVERS	gemessene Zeit (in Sekunden)
1	0,12
10	0,12
100	0,19
1 000	0,941
10 000	8,332
100 000	83,359

- Einsatz vieler Threads verlangsamt das Programm
  - Grund: Verwaltungsaufwand
- Einsatz mehrerer Threads lohnt sich, wenn die Threads immer wieder warten müssen (z.B. bei Ein-/Ausgabe)
  - ➔ Simulation durch sleep-Methode

Klasse Service: Neue Version

```
class Service implements Runnable {
   …
   public void run() {
      for(int i = start; i <= end; i++) {
         if(array[i])  result++;
         try {
              Thread.sleep(100);
         }
         catch (InterruptedException e) {  }
      }
   }
}
```

Beispiel-ergebnis:

NUMBER_OF_SERVERS	gemessene Zeit (in Sekunden)
1	10 074,230
10	1 003,593
100	101,056
1 000	10,996
10 000	13,519
100 000	90,690

Beenden von Threads

- Die Klasse **Thread** enthält eine Methode **stop()** zum Beenden von Threads
  - alle von diesem Thread gesperrten Objekte werden freigegeben
  - kann zu inkonsistenten Objektzuständen führen, wenn auf den gesperrten Objekten gerade synchronized-Methoden ausgeführt werden
  - stop()-Methode is „deprecated“ (sollte nicht verwendet werden)
- Alternative: Abbrechen ausprogrammieren
  - Verwendung eines boolean-Attributs **running**, das von außen gesetzt werden kann und das in der **run**-Methode laufend abgefragt wird

wait - notify

- Beispiel: Modellierung eines Parkhauses
  - Attribut: Anzahl der freien Parkplätze
  - Methoden: Einfahrt in das Parkhaus, Ausfahrt aus dem Parkhaus
  - Autos werden als Threads dargestellt
  - Anforderung: Einfahrt in das Parkhaus ist nicht möglich, wenn das Parkhaus voll ist

1. Lösungsversuch

```
class Parkhaus
{
   private int plätze;

   public Parkhaus(int plätze)
   {
      if(plätze < 0)
         plätze = 0;
      this.plätze = plätze;
   }

   public synchronized void passieren() // einfahren
   {
      while(plätze == 0);
      plätze--;
   }

   public synchronized void verlassen() // ausfahren
   {
      plätze++;
   }
}
```

Probleme

1. Ineffizientes aktives Warten
2. Die Lösung ist falsch
   - Sobald ein Auto in ein volles Parkhaus einfährt (Aufruf von **passieren()**), kann dieser Thread die Methode **passieren** nie mehr verlassen und damit die Sperre des Parkhaus-Objekts nie mehr freigeben
   - Verklemmung!

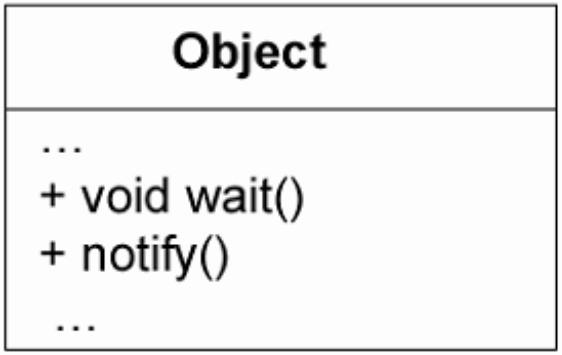
Korrekte Lösung mit wait - notify

```
class Parkhaus
{
   private int plätze;

   public Parkhaus(int plätze)
   { … wie vorher     }

   public synchronized void passieren() // einfahren
   {
      while(plätze == 0) {
         try {
              wait();
         }
         catch(InterruptedException e) { }
      }
      plätze--;
   }

   public synchronized void verlassen() // ausfahren {
      plätze++;
      notify();
   }
```

Effekt: aufrufender Thread wird blockiert und in die Warteschlange des Parkhaus-Objektes eingefügt

Effekt: ein Thread aus der Warteschlange des Parkhaus-Objektes wird entfernt

Die Methoden wait und notify



- Jedes Java-Objekt besitzt eine Sperre (lock) und eine Warteschlange für wartende Threads
- **wait** und **notify** müssen auf ein Objekt angewendet werden, das gerade vom aufrufenden Thread gesperrt ist (sonst wird eine Ausnahme geworfen)
- Die Methode **wait** blockiert den aufrufenden Thread, fügt ihn in die Warteschlange des Objekts ein und gibt die Sperre dieses Objekts frei
  - Sperren anderer Objekte werden nicht freigegeben
- Die Methode **notify** entfernt einen Thread aus der Warteschlange des Objekts; ist die Warteschlange leer, hat **notify** keine Auswirkung

Beispiel – Klasse Auto

```
class Auto extends Thread
{
   private Parkhaus parkhaus;

   public Auto(String name, Parkhaus parkhaus) {
      super(name);
      this.parkhaus = parkhaus;
      start();
   }
   public void run() {
      while(true) {  // 0 – 10 Sekunden warten
         try {
            sleep((int)(Math.random() * 10000));
         }
         catch(InterruptedException e) {  }
         parkhaus.passieren();
         System.out.println(getName()+": eingefahren");
         try { // 0 – 20 Sekunden parken
            sleep((int)(Math.random() * 20000));
         }
         catch(InterruptedException e) { }
         parkhaus.verlassen();
         System.out.println(getName()+": ausgefahren");
      } } }
```

Beispiel – main-Methode

```
public class Parkhausbetrieb
{
   public static void main(String[] args)
   {
      Parkhaus parkhaus = new Parkhaus(10);
      for(int i = 1; i <= 40; i++)
      {
         Auto a = new Auto("Auto " + i, parkhaus);
      }
   }
}
```

Vergleich der Methoden sleep und wait

- Wie die sleep-Methoden gibt es die wait-Methode auch mit einem Zeit-Parameter
  - public void wait(long millis) throws InterrupedExeption
  - befristet das Warten auf die angegebene Zeit in Millisekunden

sleep	wait
Methoden der Klasse **Thread**	Methoden der Klasse **Object**
Klassenmethoden (static)	Instanzenmethoden (non-static)
keine Bedingung für Aufruf	kann nur auf gesperrtes Objekt angewendet werden
falls Objekte gesperrt sind, bleiben sie gesperrt	Sperre des aktuellen Objekts wird aufgehoben (andere Sperren bleiben gesetzt)