

# Synchronisation

Es gibt Code-Bereiche, die nicht gleichzeitig von mehreren Threads genutzt werden dürfen. Beispielsweise wäre der gleichzeitige Zugriff auf einen Druckerspöoler nicht erwünscht.

Methoden oder Bereiche innerhalb von Methoden lassen sich synchronisieren:

```
public synchronized void ueberweisen(int betrag, int quellkonto, int zielkonto){  
    //...  
}
```

Variante a) (Java 1+):

- Methode über Schlüsselwort synchronized
- Blöcke ebenfalls über Schlüsselwort synchronized

```
public void ueberweisen(int betrag, int quellkonto, int zielkonto){  
    synchronized(new Object()){  
        //...  
    }  
}
```

Variante b) (Java 5+)

- mit Lock und Methoden lock() und unlock()

```
public void ueberweisen(int betrag, int quellkonto, int zielkonto){  
    lock.lock();  
    //...  
    lock.unlock();  
}
```

# Synchronisieren von Threads untereinander

Jeder Klasse erbt von Object die Methoden wait() und notify. Diese können in synchronisierten Blöcken genutzt werden, beispielsweise um einen Thread auf einen anderen warten zu lassen oder um Nachrichten auszutauschen.

```
Object einMonitor = new Object();
Berechner berechner = new Berechner(einMonitor);
BerechnerClient client = new BerechnerClient(einMonitor);

//Reihenfolge beachten. So wartet der Client bis der Berechner fertig
client.start();
berechner.start();
```

```
package benachrichtigen;

public class Berechner extends Thread {
    private Object monitor;

    public Berechner(Object monitor) {
        this.monitor = monitor;
    }

    public void run() {
        synchronized (monitor) {
            try {
                System.out.println("Berechner: Ich rechne gleich los");
                Thread.sleep(2000); //hier wird gerechnet
                System.out.println("Berechner: Ich bin fertig mit berechnen!");
                monitor.notify();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package benachrichtigen;

public class BerechnerClient extends Thread {
    private Object monitor;

    public BerechnerClient(Object monitor) {
        this.monitor = monitor;
    }

    public void run() {
        synchronized (monitor) {
            try {
                System.out.println("Client: Ohne die Daten vom Berechner " +
                    "kann ich nichts machen. Ich warte solange");
                monitor.wait();
                System.out.println("Client: Endlich habe ich die Ergebnisse " +
                    "und kann sie anzeigen!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

## Alternative Implementierung zu vorheriger Seite mit lock()

```
public class StartBerechnung {
    public static void main(String[] args) {
        Lock lock = new ReentrantLock();
        Condition condition = lock.newCondition();
        Berechner berechner = new Berechner(condition, lock);
        BerechnerClient client = new BerechnerClient(condition, lock);

        //Reihenfolge beachten. So wartet der Client bis der Berechner fertig ist
        client.start();
        berechner.start();
    }
}
```

```
public class Berechner extends Thread {
    private Condition condition;
    private Lock lock;

    public Berechner(Condition condition, Lock lock) {
        this.condition = condition;
        this.lock = lock;
    }

    public void run() {
        lock.lock();
        try {
            System.out.println("Berechner: Ich rechne gleich los");
            Thread.sleep(2000); //hier wird gerechnet
            System.out.println("Berechner: Ich bin fertig mit berechnen!");
            condition.signal();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        lock.unlock();
    }
}
```

```
public class BerechnerClient extends Thread {
    private Condition condition;
    private Lock lock;

    public BerechnerClient(Condition condition, Lock lock) {
        this.condition = condition;
        this.lock = lock;
    }

    public void run() {
        lock.lock();
        try {
            System.out.println("Client: Ohne die Daten vom Berechner " +
                "kann ich nichts machen. Ich warte solange");
            condition.await();
            System.out.println("Client: Endlich habe ich die Ergebnisse " +
                "und kann sie anzeigen!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        lock.unlock();
    }
}
```

# XML

Wohlgeformtheitsregeln:

- jedes XML Dokument startet mit `<?xml version="1.0" ... ?>`
- nur ein Wurzelement
- Zu jedem öffnenden ein schließendes Element
- Schachtelung aber keine Überlappung von Elementen
- Attribute in `"`
- Immer nur ein Attribut eines Namens pro Element
- ...

# Erzeugen von XML via Java

```
//Java API for XML Processing (JAXP)

//1. Dokument erzeugen
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();

Document neuesXml = db.newDocument(); //damit wird erste Zeile erzeugt
Element root = neuesXml.createElement("HTWG");
neuesXml.appendChild(root);

Element studenten = neuesXml.createElement("Studenten");
root.appendChild(studenten);

Element student = neuesXml.createElement("Student");
student.setAttribute("matrikelnummer", "1");
studenten.appendChild(student);

Element name = neuesXml.createElement("Name");
Text nadja = neuesXml.createTextNode("Nadja");
name.appendChild(nadja);
student.appendChild(name);

//2. Auf System.out ausgeben
OutputFormat format = new OutputFormat(neuesXml);
format.setIndenting(true);
XMLSerializer serializer = new XMLSerializer(System.out, format);
serializer.serialize(neuesXml);

//3. Einlesen von XML
Document unserDocument = db.parse("UnserErsteXmlDatei.xml");
Element neuesroot = unserDocument.getDocumentElement(); //WurzelElement

//to be continued
```

Vollständiger Code siehe  
"Beispielcode"

Preferences

type filter text

- General
- Ant
- AspectJ Compiler
- Checkstyle
- CodeCover
- Data Management
- GlassFish Preferences
- Groovy
- Help
- HQL editor
- Install/Update
- Java
  - Appearance
  - Build Path
  - Code Style
  - Compiler
    - Building
    - Errors/Warnings
    - Javadoc
    - Task Tags
  - Debug
  - Editor
  - Installed JREs
  - Java2Html
  - JUnit
  - Properties Files Editor
  - Visual Editor
- Java EE
- JDT Weaving
- Maven
- PL/SQL Preferences
- Plug-in Development
- Remote Systems
- Run/Debug
- Server
- Soyatec

### Errors/Warnings

[Configure Project Specific Settings...](#)

Select the severity level for the following optional Java compiler problems:

- Code style**
  - Non-static access to static member: Warning
  - Indirect access to static member: Ignore
  - Unqualified access to instance field: Ignore
  - Undocumented empty block: Ignore
  - Access to a non-accessible member of an enclosing type: Ignore
  - Method with a constructor name: Warning
  - Parameter assignment: Ignore
  - Non-externalized strings (missing/unused \$NON-NLS\$ tag): Ignore
- Potential programming problems**
- Name shadowing and conflicts**
- Deprecated and restricted API**
  - Deprecated API: Warning
    - Signal use of deprecated API inside deprecated code
    - Signal overriding or implementing deprecated method
  - Forbidden reference (access rules): Warning
  - Discouraged reference (access rules): Warning
- Unnecessary code**
- Generic types**
- Annotations**

Treat errors like fatal compiler errors (make compiled code not executable)

