Threads, Client/Server-Kommunikation und GUI-Frameworks in Java

Gero Lückemeyer

Lehrstuhl Prof. Dr. Speckenmeyer

Institut für Informatik

Universität zu Köln

Überblick

- Threads
  - Begriff
  - Realisierung in Java
- Client/Server-Kommunikation
  - Realisierung der Server-Seite in Java
  - Realisierung der Client-Seite in Java
- GUI-Frameworks Advanced Windowing Toolkit (AWT) und Swing
  - Event-Handling

Thread-Begriff

- Kontrollfluss, der mit anderen zusammen im Kontext derselben Ausführungseinheit (Prozess) ausgeführt wird, d.h.
  - schneller Wechsel und einfache Kommunikation zwischen Threads eines Prozesses
  - kein eigener Ressourcenbesitz
  - Modularisierung und Parallelität
  - Deadlock- und Race Condition-Problematik wie bei Multiprocessing

Threads - Realisierung

- Möglichkeiten:
  - Ableitung von der Klasse Thread
  - Implementierung des Interfaces Runnable
- Parameterübergabe über Konstruktor
- Interessante Methoden:
  - start() und run()
  - sleep(int iTimeout), wait(), wait (int iTimeout) und join()
  - notify() und notifyAll()

Threads - Beispiel

```
class PingPong implements Runnable {
    String sWord;
    int iDelay;
    PingPong(String sInitWord, int iInitDelay) {
        sWord=sInitWord; iDelay=iInitDelay;
    }
    public void run() {
        try {
            while(true) {
                System.out.println(sWord);
                sleep(iDelay);
            }
        } catch (InterruptedException ieCurrent) {}
    }
}
```

Threads - Aufruf

```
Runnable rnPlayerOne=new PingPong(„P1“, 33);
Runnable rnPlayerTwo=new PingPong („P2“,
   100);
Thread thPlayerOne=new Thread(rnPlayerOne);
Thread thPlayerTwo=new Thread(rnPlayerTwo);
thPlayerOne.start();
thPlayerTwo.start();
```

Deadlocks & Race Conditions

- Race Condition: mehrere Kontrollflüsse greifen unkoordiniert auf eine gemeinsame Ressource zu
- Deadlock: Menge von Kontrollflüssen, von denen jeder mindestens eine Ressource besitzt, auf die ein anderer Kontrollfluss aus der Menge wartet, und gleichzeitig auf eine Ressource wartet, die ein anderer Kontrollfluss aus der Menge besitzt

Threads, Deadlocks & Race Conditions

- Deadlocks und Race Conditions sollten gleich bei der Implementierung berücksichtigt werden
- Möglichkeiten für die nachträgliche Vermeidung von Race Conditions:
  - Objekt ableiten, entsprechende Methode als synchronized deklariert überschreiben
  - alle Zugriffe auf das Objekt in synchronized Blöcke einrahmen (fehleranfällig)

Monitore

- Monitore definieren kritische Bereiche gegenseitigen Ausschlusses
- in Java realisierbar durch das Schlüsselwort synchronized
  - in der Methodendefinition
  - mit einem Sperrobjekt um einen Block

Monitor in Java - Beispiel (1)

```
public class PCExample {
    private int iGoodsCount=0;
    private boolean bAvailable=false;
    public synchronized int consume() {
        while (bAvailable==false) {
            try {
                wait();
            } catch(InterruptedException ieCurrent) {}
        }
        bAvailable=false;
        notifyAll();
        return(iGoodsCount);
    }
```

Monitor in Java - Beispiel (2)

```
public synchronized void produce(int iValue) {
    while (bAvailable==true) {
        try {
            wait();
        } catch (InterruptedException ieCurrent) { }
    }
    iGoodsCount=iValue;
    bAvailable=true;
    notifyAll();
  }
}
```

Das synchronized Konstrukt

- (K)eine Alternative:

```
Object SperrObjekt= ... ;
synchronized (SperrObjekt) {
    Anweisungsblock
}
```

  - alle Anweisungen im Block werden ausgeführt, als seien sie synchronized deklariert
  - SperrObjekt muss innerhalb des Anweisungsblocks nicht vorkommen

Client/Server-Kommunikation

- Kommunikation über Socket-Objekte
  - datagram sockets: Abwicklung über UDP (user datagram protocol), verbindungslos, schnell aber unzuverlässig
  - stream sockets: Abwicklung über TCP (transmission control protocol), funktioniert analog zu File-I/O
- Benutzung von stream sockets für das Projekt angebracht

Server-Kommunikation

- Instanziierung eines ServerSocket-Objekts mit ServerSocket ssckGameServer=new ServerSocket (iPortNumber, int iQueueLength);
- Instanziierung eines Verbindungsobjekts mit

  Socket sckConnection=ssckGameServer.accept();
- Empfangstimeout setzen durch sckConnection.setSoTimeout(Timeout);
- Ein-/Ausgabestreams für die Verbindung anlegen, z.B. mit ObjectInputStream oisCurrent=new ObjectInputStream (sckConnection.getInputStream());
- Operationen über read() und write()
- Beenden der Verbindung über sckConnection.close();

Fehlerbehandlung

- ClassNotFoundException
- IOException und ihre Unterklassen
  - EOFException
  - SocketTimeoutException

- zuerst müssen speziellere Exceptions mit catch() aufgefangen werden, da der erste passende catch()-Block verarbeitet wird

Server-Kommunikation: Empfang (1)

```
try {
    oisCurrent=new ObjectInputStream(new BufferedInputStream
        (ClientSocket.getInputStream()));
} catch (IOException ioeCurrent) { }
while (bClientOk) {
    Object objCurrent = null;
    try {
        objCurrent = ois.readObject();
    } catch (IOException e) {break;}
    if (objCurrent != null) {
        if (! (NewObject instanceof DDWMessage)) {break;}
        iTimeoutCount=0;
        HandleDDWMessage((DDWMessage)objCurrent);
        if (!bClientOk){break;}
    }
}
```

Server-Kommunikation: Empfang (2)

- Einschluss der gesamten Sendekommunikation in folgenden Block:

```
try {
   :
} finally {
      try{
            ClientSocket.close();
      } catch (IOException ioeCurrent) { }
}
```

Server-Kommunikation: Senden (1)

```
try {
	oosCurrent = new ObjectOutputStream (sckCurrent.getOutputStream());
} catch (IOException e) {}
DDWMessage CurMessage;
while(bSendStreamOk){
	CurMessage = null;
	synchronized(MessageQueue){
		if(MessageQueue.size()>0){
			CurMessage = (DDWMessage) MessageQueue.get(0);
			MessageQueue.remove(0);
		} else {
			try{
				MessageQueue.wait();
			} catch (InterruptedException e){}
		}
	}
```

Server-Kommunikation: Senden (2)

```
if (CurMessage!=null){
    if (Client != null){
        try{
            oos.writeObject(CurMessage);
            oos.flush();
        } catch (IOException e) {}
        LastSentTime = GetCurrentMillisecond();
        if(CurMessage instanceof DDWProtocolErrorMsg){}
    }
}
}
```

Client-Kommunikation

- wie Server-Kommunikation
- allerdings direkte Instanziierung des Socket-Objekts
- nur ein Sende- und ein Empfangsthread erforderlich

Kommunikation - Alternativen

- MessageQueue
  - global
  - je Client
- ein Sende-Thread auf Server-Seite
  - für alle Clients
  - pro Client
- Session-Verwaltung und Kommunikation

GUIs mit AWT - Container

- Window
- Dialog
- Frame (speziell Applet)
- Panel (davon abgeleitet: ScrollPane)

Container Layouts

- Zuweisung an Container c mit c.setLayout()
- BorderLayout
- FlowLayout
- GridLayout

Applets - Beispiel

```
public class AppletExample extends Applet {
    Button btnCut, btnCopy, btnPaste;
    public AppletExample() {
        Panel pnButtonBar=new Panel();
        pnButtonBar.setLayout(new FlowLayout(FlowLayout.LEFT));
        btnCut=new Button(„Cut“);
        pnButtonBar.add(btnCut);
            :
        add(pnButtonBar, BorderLayout.NORTH);
    }
    public static void main(String args[]) {
        AppletExample aeFirst=new AppletExample();
    }
}
```

GUIs mit AWT - Events

- Für (fast) jede Komponente existiert ein Listener Interface, z.B.:
- Button: ActionListener
- TextField: TextListener
- List: ListSelectionListener, ListDataListener
- Mouse: MouseInputListener (bestehend aus MouseListener und MouseMotionListener)

Events - Beispiel

```
public class AppletExample extends Applet implements
    ActionListener{
     Button btnCut, btnCopy, btnPaste;
     public AppletExample() {
          Panel pnButtonBar=new Panel();
          pnButtonBar.setLayout(new FlowLayout(FlowLayout.LEFT));
          btnCut=new Button(„Cut“);
          btnCut.addActionListener(this);
            :
     }
     public void actionPerformed(ActionEvent aeCurrent) {
          if (aeCurrent.getSource()==btnCut) {...}
     }
}
```

AWT vs. Swing

- Swing-Komponenten vollständig in Java implementiert
- Nomenklatur: wie AWT-Komponenten mit vorangestelltem J
- Container von java.awt.Container abgeleitet
- Swing erst ab Java 1.2, daher keine Applets auf Plattformen ohne JRE

Grenzen von AWT

- native peer für jede Komponente auf jeder Plattform erforderlich
- kein plattformunabhängiges Look-and-Feel
- keine Transparenz
- keine GUI-Komponenten für z.B. JTree und JTabbedPane
- Swing-Erweiterungen des Event-Modells fehlen