Informatik IIb

Objektorientierte Programmierung mit Java

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Wintersemester 2010/11

Vorlesung 9. Nebenläufigkeit, Reguläre Ausdrücke

Lernziele dieser Vorlesung

Threads

Runnable

Reguläre Ausdrücke

Lernziele

- Kenntnis von Threads
- Kenntnis von Prozessablaufverwaltung
- Kenntnis Prozesssynchronisation

Nebenläufigkeit

- Nebenläufigkeit ist die Fähigkeit eines Systems, mehrere Vorgänge gleichzeitig oder zeitlich überlappend auszuführen.
- Ursprünglich ein Konzept von Betriebssystemen
- Heute auch ein Konzept von Programmiersprachen
- Nebenläufig können Threads und Prozesse sein.
- Prozess: Instrument zur Ausführung eines kompletten Programms mit eigenem Adressraum
- Threads: verschiedenen Ausführungspfade in einem Prozess mit gemeinsamem Adressraum

Anwendung von Nebenläufigkeit

- Implementierung grafischer Benutzeroberflächen
- Simulation komplexer Abläufe
- Rechenintensive Anwendungen im Hintergrund
- Durchsatzoptimierung für Daten aus der Peripherie (Netzwerk, Dateien, etc.)
- Nutzung von Mehrkernprozessoren

Nebenläufigkeit in Java

- Nebenläufiges Objekt: Klasse `Thread` des Pakets `java.lang`
- Alternativ: Interface `runnable`
- In beiden Fällen wird der Thread-Body durch die überlagerte Methode `run` zur Verfügung gestellt.
- Der Thread-Body ist der nebenläufig auszuführende Code.
- Threads können zu Gruppen zusammengefasst und priorisiert werden.

Thread erzeugen: ewiger Zähler

```
class MyThread extends Thread
{
  public void run() {
    int i = 0;
    while (true) {
      System.out.println(i++);
    }
  }
}

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

Thread erzeugen

- Klasse von `Thread` ableiten
- Methodenaufruf `start` startet den Thread
- Weitere Ausführung wird an Methode `run` übertragen
- `start` wird nach Aufruf beendet, der Aufrufer kann nebenläufig zum Thread weiterarbeiten
- Wichtig: `run` nicht direkt aufrufen, da damit kein neuer Thread erzeugt wird

Mehrere Threads erzeugen

```
public class ThreadCreate extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  public ThreadCreate() {
    super("" + ++threadCount); // Store the thread
    start();
  }
  public String toString() {
    return "#" + getName() + ":␣" + countDown;
  }
  public void run() {
    while(true) {
      System.out.println(this);
      if(--countDown == 0) return;
    }
  }
```

Mehrere Threads erzeugen (Forts.)

```
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new ThreadCreate();
  }
}
```

Thread pausieren

- Aufruf von `Thread.sleep` pausiert den aktuellen Thread.
- Zwei Varianten
  - `public static void sleep(long millis)`
    Unterbrechung in Millisekunden
  - `public static void sleep(long millis, int nanos)`
    Unterbrechung in Millisekunden und Nanosekunden
- `sleep` erfordert einen try-catch-Block, weil `sleep` unterbrochen werden kann, bevor die Zeit abgelaufen ist (Ausnahme vom Typ `InterruptedException`).

Thread unterbrechen

- Durch Aufruf von `interrupt` wird ein Flag gesetzt, das eine Unterbrechungsanforderung signalisiert.
- Durch Aufruf von `isInterrupted` kann der Thread feststellen, ob das Abbruchflag gesetzt wurde und der Thread beendet werden soll.
- Die statische Methode `interrupted` stellt den Status des Abbruchflags beim aktuellen Thread fest.
- Entspricht dem Aufruf von currentThread().isInterrupted(), setzt aber zusätzlich das Abbruchflag auf seinen initialen Wert false zurück.

Thread unterbrechen (Forts.)

```
public class ThreadInterrupt extends Thread
{
  int cnt = 0;

  public void run()
  {
    while (true) {
      if (isInterrupted()) {
        break;
      }
      printLine(++cnt);
    }
  }
```

Thread unterbrechen (Forts.)

```
private void printLine(int cnt)
{
  //Zeile ausgeben
  System.out.print(cnt + ":␣");
  for (int i = 0; i < 30; ++i) {
    System.out.print(i == cnt % 30 ? "*␣" : ".␣");
  }
  System.out.println();
  //100 ms. warten
  try {
    Thread.sleep(100);
  } catch (InterruptedException e) {
    interrupt();
  }
}
```

Thread unterbrechen (Forts.)

```
  public static void main(String[] args)
  {
  ThreadInterrupt th = new ThreadInterrupt();
    {
      //Thread starten
      th.start();
      //3 Sekunden warten
      try {
        Thread.sleep(3000);
      } catch (InterruptedException e) {
      }
      //Thread unterbrechen
      th.interrupt();
    }
  }
}
```

join

- Die Methode `join` wartet auf das Ende des Threads, für den sie aufgerufen wurde.
- Falls ein Thread `t.join()` für einen anderen Thread `t` aufruft, wartet der aufrufende Thread, bis Thread `t` beendet ist.
- Ob ein Thread `t` noch lebt, ist mit der Methode `t.isAlive` feststellbar.
- Ein try-catch-Block ist erforderlich, weil `join` durch ein `interrupt` für den aufrufenden Thread beendet werden kann.

Sleeper Joiner

- Ein Sleeper ist ein Thread, der für eine bestimmte Zeit wartet (spezifizert im Konstruktor)
- Der Aufruf von `sleep` in `run` kann terminieren, wenn die Zeit abgelaufen ist, oder unterbrochen werden.
- Im catch-Block wird ausgegeben, ob eine Unterbrechung statt fand.
- Ein Joiner ist ein Thread, der auf das Aufwachen eines Sleepers wartet (durch Aufruf von `join` für den Sleeper).
- Beispiel: Mehrere Sleeper mit jeweils einem Joiner, Ausgabe des Verhaltens (unterbrochen, aufgewacht)
- Der Joiner beendet sich zusammen mit seinem Sleeper.

Sleeper Joiner (Forts.)

```
class Sleeper extends Thread {
  private int duration;
  public Sleeper(String name, int sleepTime) {
    super(name);
    duration = sleepTime;
    start();
  }
```

Sleeper Joiner (Forts.)

```
  public void run() {
    try {
      sleep(duration);
    } catch (InterruptedException e) {
      System.out.println(getName() +
       "␣was␣interrupted.␣" +
       "isInterrupted():␣" + isInterrupted());
      return;
    }
    System.out.println(getName() +
     "␣has␣awakened");
  }
}
```

Sleeper Joiner (Forts.)

```
class SleeperJoiner extends Thread {
  private Sleeper sleeper;
  public SleeperJoiner(String name,
   Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();
  }
```

Sleeper Joiner (Forts.)

```
  public void run() {
   try {
      sleeper.join();
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
    System.out.println(getName() +
     "␣join␣completed");
  }
}
```

Sleeper Joiner (Forts.)

```
public class Joining {
  public static void main(String[] args) {
    Sleeper
      sleepy = new Sleeper("Sleepy", 1500),
      grumpy = new Sleeper("Grumpy", 1500);
    SleeperJoiner
      dopey = new SleeperJoiner("Dopey", sleepy),
      doc = new SleeperJoiner("Doc", grumpy);
    grumpy.interrupt();
  }
}
```

Thread über Interface erzeugen

- Soll eine Klasse, die nicht von Thread abgeleitet ist, als Thread laufen, muss das Interface `runnable` verwendet werden.
- Vorgehen

  1. Zunächst wird ein neues Thread-Objekt erzeugt.
  2. An den Konstruktor wird das Objekt übergeben, das nebenläufig ausgeführt werden soll.
  3. Die Methode `start` des neuen Thread-Objekts wird aufgerufen.

- Bemerkung: `Thread` implementiert `runnable`.

Thread über Interface erzeugen (Forts.)

```
public class RunnableThread implements Runnable {
  private int countDown = 5;
  public String toString() {
    return "#" + Thread.currentThread().getName() +
      ": " + countDown;
  }
  public void run() {
    while(true) {
      System.out.println(this);
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 5; i++)
      new Thread(new RunnableThread(), "" + i).start();
  }
}
```

Reguläre Ausdrücke (Regular Expressions)

- Reguläre Ausdrücke beschreiben Mengen von Zeichenketten, die gemeinsame Charakteristiken haben.
- Sie werden benutzt, um Text zu suchen, zu bearbeiten und zu manipulieren.
- Reguläre Ausdrücke werden in einer speziellen Syntax beschrieben.
- Reguläre Ausdrücke gibt es in verschiedenen Programmiersprachen: z. B. grep, Perl, Tcl, Python, PHP, und awk.
- Reguläre Ausdrücke sind äquivalent zu endlichen deterministischen Automaten

Reguläre Ausdrücke in Java

- Die Benutzung und Syntax ist der von Perl sehr ähnlich.
- Definiert in drei Klassen im Paket `java.util.regex`
- Ein `Pattern`-Objekt ist eine kompilierte Repräsentation eines regulären Ausdrucks. Erzeugung mit der statischen Methode `compile`.
- ein `Matcher`-Objekt ist eine Maschine, die Matching-Operationen des `Pattern` auf einem `String` ausführt.
- Ein `PatternSyntaxException`-Objekt zeigt einen Syntaxfehler eines regulären Ausdrucks an (im `Pattern`).

Suchen von Strings-Literalen

- regex: `foo`
- Suchstring: `foo`

```
Pattern pattern =
            Pattern.compile("foo");
Matcher matcher =
            pattern.matcher("foo");
while (matcher.find()) {
  System.out.format(
  "I␣found␣the␣text␣\"%s\"␣starting␣at␣" +
  "index␣%d␣and␣ending␣at␣index␣%d.%n",
  matcher.group(), matcher.start(),
  matcher.end());
}
// I found the text "foo" starting at index 0
//  and ending at index 3.
```

Metacharacters

- Manche Zeichen haben in regulären Ausdrücken eine spezielle Bedeutung.
- regex: `cat.`
- Suchstring: cats
- `I found the text "cats"starting at index 0 and ending at index 4.`
- Der Punkt (`.`) stehr für ein beliebiges Zeichen.
- Alle Metacharacters: `([{\^-$|]})?*+.`
- Nicht immer haben diese Zeichen ein spezielle Bedeutung.
- Soll ein Zeichen nicht als Metacharacter interpretiert werden, muss ein \ davor gestellt werden.

Bedeutung von Metcharacters

Zeichenklasse	
`[abc]`	a, b oder c
`[^abc]`	jedes Zeichen außer a, b oder c (Negation)
`[a-zA-Z]`	a bis z oder A bis Z (Bereich)
`[a-d[m-p]]`	a bis d oder m bis p, das selbe wie `[a-dm-p]` (Vereinigung)
`[a-z&&[def]]`	d, e, oder f (Schnittmenge)
`[a-z&&[^bc]]`	a bis z ohne b und c, das selbe wie `[ad-z]` (Differenz)
`[a-z&&[^m-p]]`	a bis z ohne m bis p, das selbe wie `[a-lq-z]` (Differenz)

- regex: `[0-9&&[^345]]`
- Suchstring: 3
- `No match found.`

Vordefinierte Zeichenklassen

Zeichenklasse	
.	beliebiges Zeichen (auch Zeilenende)
\d	[0-9] (Ziffer)
\D	[^0-9] (nicht-Ziffer)
\s	[\t\n\x0B\f\r] (Whitespace)
\S	[^\s] (nicht-Whitespace)
\w	[a-zA-Z_0-9] (Wortzeichen)
\W	[^\w] (nicht-Wortzeichen)

- regex: \D
- Suchstring: 3
- No match found.
- regex: \d
- Suchstring: 3
- I found the text "3"starting at index 0 and ending at index 1.

Quantifizierer

Mit Quantifizierern kann die Anzahl des Auftretens von Suchstrings spezifiziert werden.

Quantifizierer			Bedeutung
greedy	reluctant	possessive	
`X?`	`X??`	`X?+`	`X` ein- oder keinmal
`X*`	`X*?`	`X*+`	`X` kein- oder mehrmal
`X+`	`X+?`	`X++`	`X` ein- oder mehrmal
`X{n}`	`X{n}?`	`X{n}+`	`X` genau n-mal
`X{n,}`	`X{n,}?`	`X{n,}+`	`X` mindestens n-mal
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	`X` mindestens n-, höchstens m-mal

Beispiel greedy Quantifizierer

- regex: a?
- Suchstring: aaa
-
```
I found the text "a" starting at index 0 and ending at index 1.
I found the text "a" starting at index 1 and ending at index 2.
I found the text "" starting at index 2 and ending at index 3.
```
- regex: a*
- Suchstring: aaa
-
```
I found the text "aaa" starting at index 0 and ending at index 3.
I found the text "" starting at index 2 and ending at index 3.
```
- regex: a+
- Suchstring: aaa
-
```
I found the text "aaa" starting at index 0 and ending at index 3.
```