Grundlagen der Programmierung 2 SS 2005 - Lösung 5

Lösung zu Aufgabe 16

Zu dieser Aufgabe gibt es keine Musterlösung.

Lösung zu Aufgabe 17

Zu dieser Aufgabe gibt es keine Musterlösung.

Lösung zu Aufgabe 18

a)

Ein Prozess (schwergewichtig) besitzt einen eigenen Adressraum. Prozesse können sich nicht gegenseitig beeinflussen, sondern besitzen ihren Adressraum und ihre Ressourcen exklusiv.

Jeder Prozess besitzt einen oder mehrere Threads. Ein Thread (leichtgewichtig) ist ein Teilprozess. Er kann nicht ohne seinen Mutterprozess existieren.
Mehrere Threads teilen sich ihren Prozess und damit dessen Ressourcen und Adressraum. Sie sind aber eigenständig innerhalb eines Prozesses.

b)

In Java repräsentieren Objekte der Klasse Thread leichtgewichtige Prozesso (*Threads*) im Sinne von Teilaufgabe (a). Die gesamte Java-Laufzeitumgebung, wie sie vom Kommando `java` erzeugt wird, erscheint gegegnüber dem Betriebssystem des Rechners als ein schwergewichtiger Prozess. Darin wird, wenn sonst nichts anderes spezifiziert ist, für den Programmierer unsichtbar zunächst ein Thread erzeugt, der die `main`-Methode des Java-Programms ausführt. Das zugehörige Thread-Objekt kann z.B. mit der Klassenmethode `currentThread()` innerhalb der `main`-Methode abgefragt werden.

Java erlaubt es zudem, mehrere Thread-Objekte zu erzeugen und als Teilprozesse unabhängig voneinander ablaufen zu lassen. Auf Rechnern mit mehreren Prozessoren werden diese echt parallel ausgeführt. Gibt es mehr Ausführungsstränge als Prozessoren findet eine verzahnte, nur scheinbar parallele Ausführung statt.

Threads besitzen Prioritäten, die jedoch nur eine sehr grobe Einflussnahme auf die Ausführungsreihenfolge der Threads ermöglichen und sich insbesondere nicht zur Koordination mehrerer Threads eignen. Die dafür passenden Mechanismen werden im Kapitel 12 der Vorlesung vorgestellt.

Im Falle eines Applets startet der Browser innerhalb seines Prozesses einen Thread, der das Applet ausführt.

c)

`start`, `run` und `stop`:

Erst durch Aufruf der Objektmethode `start` der Klasse `Thread` wird ein neuer Thread in die Lage versetzt, demnächst unabhängig ablaufen zu können. Er bekommt den Zustand *alive* (lebendig) und seine `run`-Methode wird implizit aufgerufen. Wenn diese beendet wird, ist auch der Zustand *alive* beendet.

Ein toter Thread kann nicht mehr durch erneuten Aufruf von `start` neu gestartet werden, in diesem Fall muss ein neues Thread-Objekt werden. Die `start`-Methode schafft die Voraussetzungen zum Ablauf des Threads. Der eigentliche Ablauf des Threads gehört in dessen `run`-Methode.

Der natürliche Tod eines Threads besteht immer aus der Beendigung der `run`-Methode. Der gewaltsame Tod eines Threads kann durch Aufruf der Objektmethode `stop` der Klasse Thread "von außen" erzwungen werden. Beim Aufruf von `stop` ist aber im Allgemeinen unbekannt, was der Thread gerade tut. Für den Thread sieht es so aus, als wäre spontan eine Exception der Klasse `ThreadDeath` aufgetreten. Da es

Programmteile innerhalb von Threads gibt, die keinesfalls unterbrochen werden dürfen, die `stop`-Methode darauf aber keine Rücksicht nimmt, sollte `stop` nicht mehr verwendet werden. Stattdessen sollte man Mechanismen zum kurzfristigen, aber kontrollierten Beenden eines Threads selber einbauen.

Durch Aufruf von `run` wird kein neuer Ausführungsstrang gestartet, sondern die Kontrolle geht an den Anfang der Methode über, der ursprüngliche Thread führt die Aktionen aus und kehrt zurück. Im Falle von `start` hingegen wird ein zweiter Ausführungsstrang gestartet und danach arbeitet der ursprüngliche Thread sofort nach dem `start`-Aufruf weiter, während der neu erzeugte parallel dazu Thread die `run`-Methode abarbeitet.

Grundsätzlich passiert also beim direkten Aufruf von `run` fast genau das Gleiche wie beim Aufruf von `start`, nur mit dem Unterschied, dass die Objektmethode `run` nicht parallel zum übrigen Programm abgearbeitet wird. Der aktuelle Thread bearbeitet die `run`-Methode sequenziell, bis sie zu Ende ist und die Anweisungen nach dem Aufruf an die Reihe kommen. Dieser Fehler fällt nicht immer direkt auf, denn die Aktionen in `run` finden ja statt - nur eben nicht nebenläufig.

d)

Durch Aufruf der Klassenmethode `Thread.currentThread()` der Klasse `Thread` bekommt man eine Referenz auf das gerade aktive Thread-Objekt. In manchen Situationen (z.B. bei der Ereignisbehandlung oder beim Laden von Grafiken) erzeugen Methoden aus der Java-Bibliothek intern weitere Threads, auf die es im eigenen Programm keine Referenz gibt. Auf diese Thread-Objekte kann man dann per `Thread.currentThread()` zugreifen.

Mit der Objektmethode `isAlive` der `Klasse` Thread kann der Zustand alive eines Threads abgefragt werden. Dadurch kann sich ein "Mutter-Thread" darüber informieren, ob von ihm erzeugte "Töchter-Threads" noch leben, d.h. ihre run-Methode noch nicht vollständig abgearbeitet wurde.

Lösung zu Aufgabe 19

Die ursprüngliche Lösung wurde so modifiziert, dass nun `JApplet` und nicht mehr `JFrame` erweitert wird. Dazu werden die grafischen Komponenten nun in der `init`-Methode und nicht mehr im Konstruktor erzeugt.

In der `init`-Methode werden über zwei Parameter die Anzahl der Zeilen und Spalten des Spielfelds aus der HTML-Datei eingelesen und die Anzahl der Felder berechnet. Falls die Anzahl der Felder ungerade ist, so reduzieren wir die Anzahl der Felder um eins. Der Codeteil dazu sieht so aus:

```
// Objektvariablen für das Spiel
private int rows, columns, fields;

public void init()
{  // init()-Methode statt Konstruktor

   // Nur ein Listener-Objekt fuer alle Felder:
   MemoryListener memListener = new MemoryListener();
   rows = Integer.parseInt (getParameter ("rows"));
   columns = Integer.parseInt (getParameter ("columns"));
   fields = rows*columns;
   Container content = this.getContentPane();
   content.setLayout(new GridLayout(rows, columns));
   boolean isOdd = fields % 2 == 1;
   // letztes Feld löschen
   if (isOdd) fields--;

   // Buttons fuer die Felder erzeugen:
   buttons = new CharButton [fields];
   initLabels ();
   for (int i = 0; i < fields; i++)
   {
      buttons [i] = new CharButton (getCharacter ());
```

```
      content.add(buttons [i]);
      buttons [i].addActionListener (memListener);
      buttons [i].cover();
   }
}
```

Zudem beschriften wir die Felder mit Integer-Werten statt mit Buchstaben:

```
// Die Beschriftungen der Felder:
private String[] labels;
private int labelsLength;

private void initLabels ()
{
   int cnt = 0;
   labels = new String[fields];
   labelsLength = fields;
   for (int i = 0; i < fields; i+=2)
   {
      labels [i] = labels [i+1] = (new Integer (cnt)).toString();
      cnt++;
   }
}
```

Der Rest des Quellcodes bleibt unverändert bis auf die Anpassungen hinsichtlich `JApplet`.

Hier befinden sich die Java-Datei `Memory_LSG.java` und die dazugehörige HTML-Datei `Memory.html`, um das Applet kompilieren und testen zu können.