Script generated by TTT

Title: Seidl: Informatik_1 (14.01.2013)

Date: Mon Jan 14 17:13:27 CET 2013

Duration: 90:40 min

Pages: 40

20.1 Futures

- Die Berechnung eines Zwischenergebnisses kann lange dauern.
- Während dieser Berechnung kann möglicherweise etwas anderes sinnvolles berechnet werden.

20.1 Futures

- Die Berechnung eines Zwischenergebnisses kann lange dauern.
- Während dieser Berechnung kann möglicherweise etwas anderes sinnvolles berechnet werden.

Idee:

- Berechne das Zwischenergebnisses in einem eigenen Thread.
- Greife auf den Wert erst zu, wenn sich der Thread beendet hat.

$\Longrightarrow$ Futures

Eine Future startet die Berechnung eines Werts, auf den später zugegriffen wird ...

Das generische Interface

```
public interface Callable<T> {
   T call throws Exception ();
}
```

aus `java.util.concurrency` beschreibt Klassen, für deren Objekte ein Wert vom Typ `T` berechnet werden kann.

Eine Future startet die Berechnung eines Werts, auf den später zugegriffen wird ...

Das generische Interface

```
public interface Callable<T> {
   T call throws Exception ();
}
```

aus java.util.concurrency beschreibt Klassen, für deren Objekte ein Wert vom Typ T berechnet werden kann. Wir implementieren:

```
public class Future<T> implements Runnable {
   private T value = null;
   private Exception exc = null;
   private Callable<T> work;
   private Thread task;
...
```



```
...
public Future<T>(Callable<T> w) {
   work = w;
   task = new Thread (this);
   task.start();
   }
public void run() {
   try {value = work.call();}
   catch (Exception e) { exc = e;}
   }
public T get() throws Exception {
   task.join();
   if (value == null) throw exc;
   return value;
   }
}
```


Eine Future startet die Berechnung eines Werts, auf den später zugegriffen wird ...

Das generische Interface

```
public interface Callable<T> {
   T call throws Exception ();
}
```

aus java.util.concurrency beschreibt Klassen, für deren Objekte ein Wert vom Typ T berechnet werden kann. Wir implementieren:

```
public class Future<T> implements Runnable {
   private T value = null;
   private Exception exc = null;
   private Callable<T> work;
   private Thread task;
...
```

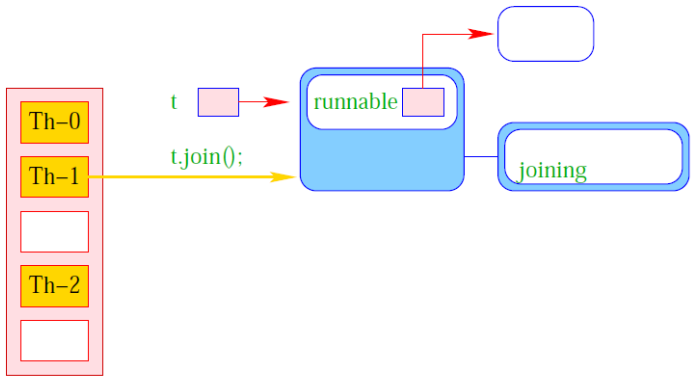
730

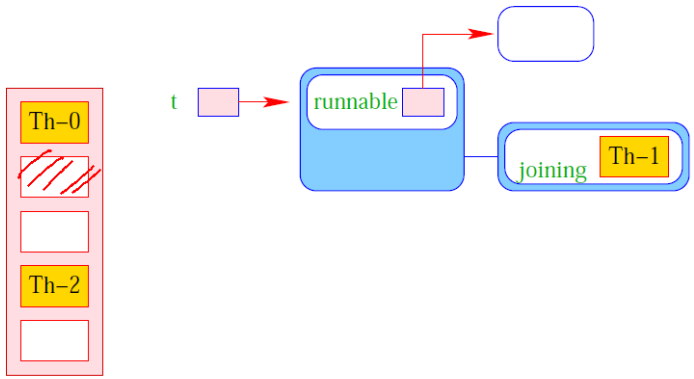
```
...
public Future<T>(Callable<T> w) {
   work = w;
   task = new Thread (this);
   task.start();
   }
public void run() {
   try {value = work.call();}
   catch (Exception e) { exc = e;}
   }
public T get() throws Exception {
   task.join();
   if (value == null) throw exc;
   return value;
   }
}
```

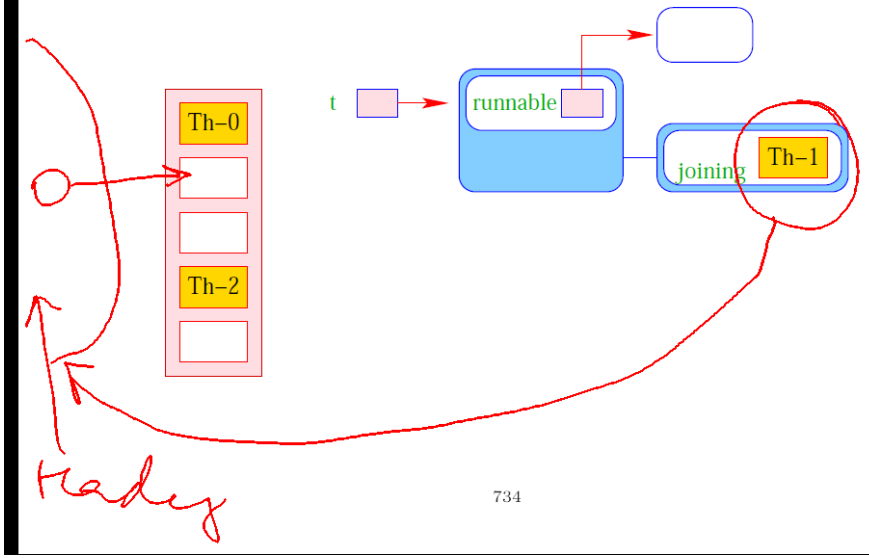
731

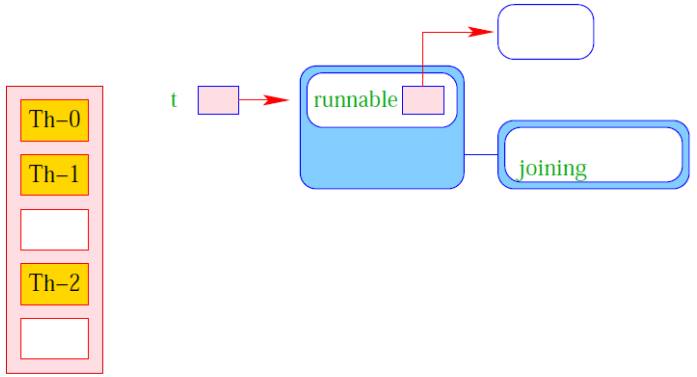
```
...
public Future<T>(Callable<T> w) {
   work = w;
   task = new Thread (this);
   task.start();
   }
public void run() {
   try {value = work.call();}
   catch (Exception e) { exc = e;}
   }
public T get() throws Exception {
   task.join();
   if (value == null) throw exc;
   return value;
   }
}
```


- Der Konstruktor erhält ein `Callable`-Objekt.
- Die Methode `run()` ruft für dieses Objekt die Methode `call()` auf und speichert deren Ergebnis in dem Attribut `value` — bzw. eine geworfene Exception in `exc` ab.
- Der Konstruktor legt ein Thread-Objekt für die Future an und startet diesen Thread, der dann `run()` ausführt.
- Die Methode `get()` wartet auf Beendigung des Threads. Dazu verwendet sie die Objekt-Methode `public final void join() throws InterruptedException` der Klasse `Thread` ...
- Dann liefert `get()` den berechneten Wert zurück — falls dieser nicht `null` ist. Andernfalls wird die Exception `exc` geworfen.

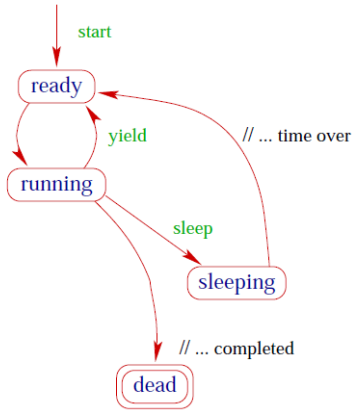




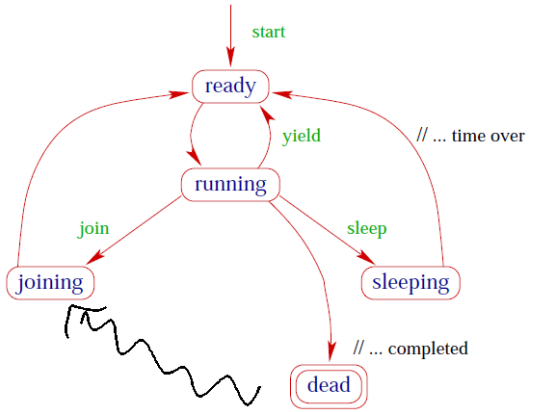




Erweitertes Zustandsdiagramm:



Erweitertes Zustandsdiagramm:



... anderes Beispiel:

```
public class Join implements Runnable {
   private static int count = 0;
   private int n = count++;
   private static Thread[] task = new Thread[3];
   public void run() {
      try {
         if (n>0) {
            task[n-1].join();
            System.out.println("Thread-"+n+" joined Thread-"+(n-1));
         }
      } catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   } ...
```



```
   ...
   public static void main(String[] args) {
      for(int i=0; i<3; i++)
         task[i] = new Thread(new Join());
      for(int i=0; i<3; i++)
         task[i].start();
   }
}  // end of class Join
```

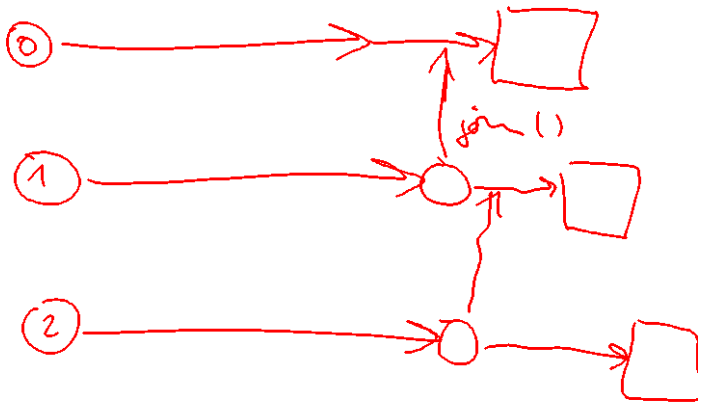
... liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```

Spaßeshalber betrachten wir noch eine kleine Variation des letzten Programms:

... anderes Beispiel:

```
public class Join implements Runnable {
   private static int count = 0;
   private int n = count++;
   private static Thread[] task = new Thread[3];
   public void run() {
      try {
         if (n>0) {
            task[n-1].join();
            System.out.println("Thread-"+n+" joined Thread-"+(n-1));
         }
      } catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   } ...
```

```
   ...
   public static void main(String[] args) {
      for(int i=0; i<3; i++)
         task[i] = new Thread(new Join());
      for(int i=0; i<3; i++)
         task[i].start();
   }
} // end of class Join
```

... liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```

Spaßeshalber betrachten wir noch eine kleine Variation des letzten Programms:


```
   ...
   public static void main(String[] args) {
      for(int i=0; i<3; i++)
         task[i] = new Thread(new Join());
      for(int i=0; i<3; i++)
         task[i].start();
   }
} // end of class Join
```

... liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```

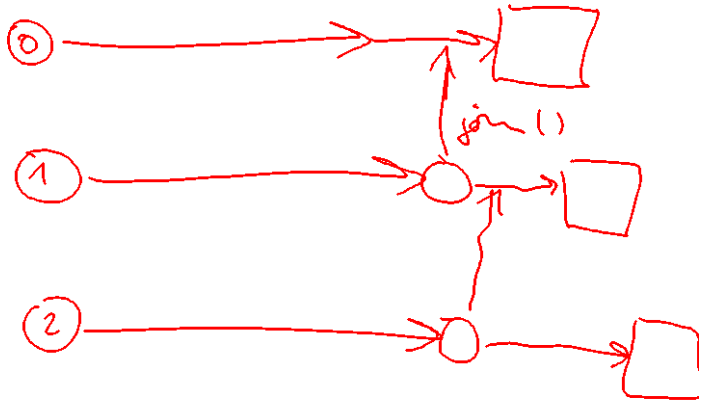
Spaßeshalber betrachten wir noch eine kleine Variation des letzten Programms:

... anderes Beispiel:

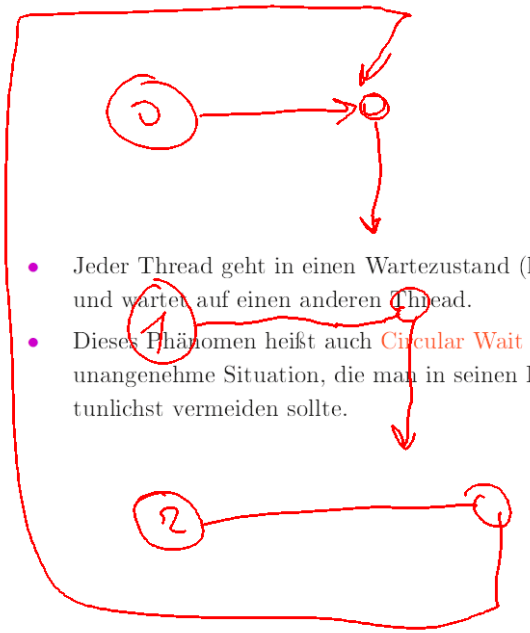
```
public class Join implements Runnable {
   private static int count = 0;
   private int n = count++;
   private static Thread[] task = new Thread[3];
   public void run() {
      try {
         if (n>0) {
            task[n-1].join();
            System.out.println("Thread-"+n+" joined Thread-"+(n-1));
         }
      } catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   } ...
```



```
public class CW implements Runnable {
   private static int count = 0;
   private int n = count++;
   private static Thread[] task = new Thread[3];
   public void run() {
      try { task[(n+1)%3].join(); }
      catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   }
   public static void main(String[] args) {
      for(int i=0; i<3; i++)
         task[i] = new Thread(new CW());
      for(int i=0; i<3; i++) task[i].start();
   }
} // end of class CW
```




```
public class CW implements Runnable {
   private static int count = 0;
   private int n = count++;
   private static Thread[] task = new Thread[3];
   public void run() {
      try { task[(n+1)%3].join(); }
      catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   }
   public static void main(String[] args) {
      for(int i=0; i<3; i++)
         task[i] = new Thread(new CW());
      for(int i=0; i<3; i++) task[i].start();
   }
} // end of class CW
```

- Jeder Thread geht in einen Wartezustand (hier: joining) über und wartet auf einen anderen Thread.
- Dieses Phänomen heißt auch Circular Wait oder Deadlock – eine unangenehme Situation, die man in seinen Programmen tunlichst vermeiden sollte.

- Jeder Thread geht in einen Wartezustand (hier: joining) über und wartet auf einen anderen Thread.
- Dieses Phänomen heißt auch Circular Wait oder Deadlock – eine unangenehme Situation, die man in seinen Programmen tunlichst vermeiden sollte.

20.2 Monitore

- Damit Threads sinnvoll miteinander kooperieren können, müssen sie miteinander Daten austauschen.
- Zugriff mehrerer Threads auf eine gemeinsame Variable ist problematisch, weil nicht feststeht, in welcher Reihenfolge die Threads auf die Variable zugreifen.
- Ein Hilfsmittel, um geordnete Zugriffe zu garantieren, sind Monitore.

... ein Beispiel:


```
public class Inc implements Runnable {
   private static int x = 0;
   private static void pause(int t) {
      try {
         Thread.sleep((int) (Math.random()*t*1000));
      } catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   }
   public void run() {
      String s = Thread.currentThread().getName();
      pause(3); int y = x;
      System.out.println(s+ " read "+y);
      pause(4); x = y+1;
      System.out.println(s+ " wrote "+(y+1));
   }
```



```
      ...
      public static void main(String[] args) {
         (new Thread(new Inc())).start();
         pause(2);
         (new Thread(new Inc())).start();
         pause(2);
         (new Thread(new Inc())).start();
      }
   }  // end of class Inc
```

- `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- `public final String getName();` liefert den Namen des Thread-Objekts.
- Das Programm legt für drei Objekte der Klasse `Inc` Threads an.
- Die Methode `run()` inkrementiert die Klassen-Variable `x`.

```
public class Inc implements Runnable {
   private static int x = 0;
   private static void pause(int t) {
      try {
         Thread.sleep((int) (Math.random()*t*1000));
      } catch (InterruptedException e) {
         System.err.println(e.toString());
      }
   }
   public void run() {
      String s = Thread.currentThread().getName();
      pause(3); int y = x;
      System.out.println(s+ " read "+y);
      pause(4); x = y+1;
      System.out.println(s+ " wrote "+(y+1));
   }
```



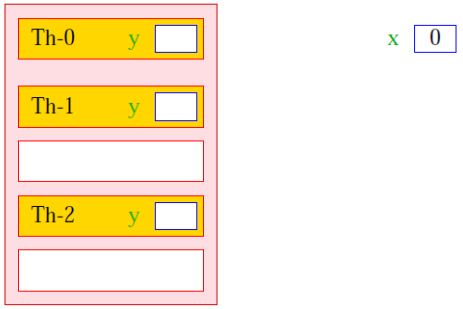
```
   ...
   public static void main(String[] args) {
      (new Thread(new Inc())).start();
      pause(2);
      (new Thread(new Inc())).start();
      pause(2);
      (new Thread(new Inc())).start();
   }
} // end of class Inc
```

- `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- `public final String getName();` liefert den Namen des Thread-Objekts.
- Das Programm legt für drei Objekte der Klasse `Inc` Threads an.
- Die Methode `run()` inkrementiert die Klassen-Variable `x`.

Die Ausführung liefert z.B.:

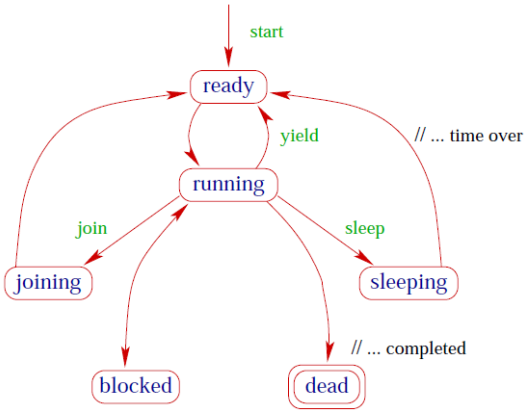
```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

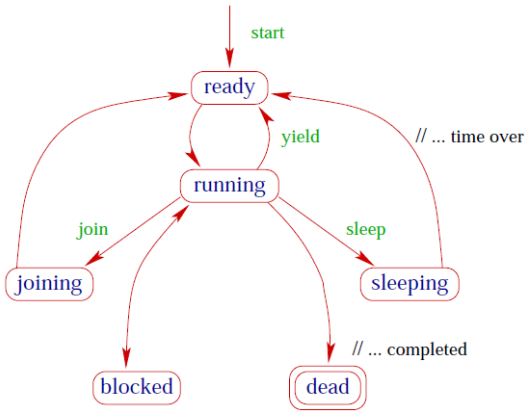
Der Grund:



Idee:

- Inkrementieren der Variable `x` sollte ein atomarer Schritt sein, d.h. nicht von parallel laufenden Threads unterbrochen werden können.
- Mithilfe des Schlüsselworts `synchronized` kennzeichnen wir Objekt-Methoden einer Klasse L als ununterbrechbar.
- Für jedes Objekt obj der Klasse L kann zu jedem Zeitpunkt nur ein Aufruf `obj.synchMeth(...)` einer `synchronized`-Methode `synchMeth()` ausgeführt werden. Die Ausführung einer solchen Methode nennt man kritischen Abschnitt ("critical section") für die gemeinsame Resource obj.
- Wollen mehrere Threads gleichzeitig in ihren kritischen Abschnitt für das Objekt obj eintreten, werden alle bis auf einen blockiert.





- Jedes Objekt obj mit `synchronized`-Methoden verfügt über:
  1. über ein boolesches Flag `boolean locked;` sowie
  2. über eine Warteschlange `ThreadQueue blockedThreads`.
- Vor Betreten seines kritischen Abschnitts führt ein Thread (implizit) die atomare Operation `obj.lock()` aus:

```
private void lock() {
   if (!locked) locked = true; // betrete krit. Abschnitt
   else {                      // Lock bereits vergeben
      Thread t = Thread.currentThread();
      blockedThreads.enqueue(t);
      t.state = blocked;       // blockiere
   }
}  // end of lock()
```

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
private void unlock() {
   if (blockedThreads.empty())
      locked = false; // Lock frei geben
   else {             // Lock weiterreichen
      Thread t = blockedThreads.dequeue();
      t.state = ready;
   }
}  // end of unlock()
```

- Dieses Konzept nennt man Monitor.