Informatik B

Vorlesung 7

Threads

Rückblick

- WordCount
  - **`Filter`**
  - Systemproperties
  - Umgebungsvariablen
- Klasse **`String`**
  - Strings verketten
  - Strings teilen
  - Formatierte Ausgabe
  - Variable Parameterliste

Nebenläufigkeit

- Verschiedene Programme werden scheinbar gleichzeitig ausgeführt (Multitaskingfähigkeit)
- Quasiparallelität, Nebenläufigkeit
- Das Betriebssystem gewährleistet, das auf Einprozessormaschinen die Prozesse alle paar Millisekunden umgeschaltet werden
- Das Programm ist nicht wirklich parallel, das Betriebssystem gaukelt dies durch abwechselnde Bearbeitung der Prozesse vor
- Wenn mehrere Prozessoren oder mehrere Prozessor-Kerne am Werke sind, werden die Programmteile tatsächlich parallel abgearbeitet
- Die Umschaltung übernimmt der Scheduler

Nebenläufigkeit

- Die dem Betriebssystem bekannten aktiven Programme bestehen aus Prozessen
- Ein Prozess setzt sich aus dem Programmcode und den Daten zusammen und besitzt einen eigenen Adressraum
- Ressourcen wie geöffnete Dateien oder belegte Schnittstellen gehören dazu
- Die virtuelle Speicherverwaltung des Betriebssystems trennt die Adressräume der einzelnen Prozesse
- So ist es nicht möglich, dass ein Prozess den Speicherraum eines anderen Prozesses korrumpiert; er sieht den anderen Speicherbereich nicht
- Damit Prozesse untereinander Daten austauschen können, wird ein besonderer Speicherbereich als Shared-Memory markiert

Threads und Prozesse

- Bei modernen Betriebssystemen gehört zu jedem Prozess mindestens ein Thread, der den Programmcode ausführt
- Innerhalb eines Prozesses kann es mehrere Threads geben, die alle zusammen in demselben Adressraum ablaufen
- Threads eines Prozesses können untereinander auf ihre öffentlichen Daten zugreifen
- Die Programmierung von Threads ist in Java einfach möglich, und die quasi parallel ablaufenden Aktivitäten ergeben für den Benutzer den Eindruck von Gleichzeitigkeit
- In Java ist auch multithreaded Software möglich, wenn das Betriebssystem des Rechners keine Threads direkt verwendet
- In diesem Fall simuliert die virtuelle Maschine die Parallelität, indem sie die Synchronisation und die verzahnte Ausführung regelt

Threads und Prozesse

- Unterstützt das Betriebssystem Threads direkt, bildet die JVM die Thread-Verwaltung in der Regel auf das Betriebssystem ab (nativen Threads)
- Ob die Laufzeitumgebung nativ Threads nutzt oder nicht, steht nicht in der Spezifikation der JVM
- Die JVM garantiert die korrekt verzahnte Ausführung
- Hier können Probleme auftreten, die z.B. auch von Datenbanken (Transaktionen) her bekannt sind
- Es besteht die Gefahr konkurrierender Zugriffe auf gemeinsam genutzte Ressourcen
- Um dies zu vermeiden, kann der Programmierer durch synchronisierte Programmblöcke gegenseitigen Ausschluss sicherstellen
- Dadurch steigt die Gefahr von Verklemmungen (*deadlocks*)

Einsatzgebiete von Threads

- Vermeidung von Blockaden der Benutzerschnittstelle, mittels Threads kann eine Anwendung weiter auf Eingaben reagieren
- Ereignisse nach einer bestimmten Zeit ausführen, z.B. eine Diashow
- Parallelisierung einer Anwendung auf mehrere Prozessoren
- Implementierung blockierender Schnittstellen, z.B. Netzwerkprogrammierung

Thread

- Für nebenläufige Programme in Java ist die Klasse **`Thread`** zuständig
- Jeder laufende Thread ist ein Objekt dieser Klasse
- Threads können auf zwei Arten erzeugt werden:
  - Implementierung des Interface **`Runnable`**
  - Vererbung der Klasse **`Thread`** (die das Interface **`java.lang.Runnable`** implementiert)
- Die Methode **`run()`** aus dem Interface **`Runnable`** muss implementiert werden, bzw. muss von der Klasse **`Thread`** überschrieben werden
- Die Methode **`run()`** enthält die auszuführenden Anweisungen

Thread-Status

- Ein Thread kann sich in sechs verschiedenen Stadien befinden:
  - **NEW**: Thread wurde erstellt, aber noch nicht gestartet
  - **RUNNABLE**: Der Thread wurde gestartet
  - **BLOCKED**: Der Thread läuft momentan nicht, sondern wartet auf einen Monitor
  - **WAITING**: Der Thread wartet aufgrund eines Methodenaufrufs **`wait()`** oder **`join()`**
  - **TIMED_WAITING**: Der Thread waret aufgrund der Methode **`sleep()`** oder aufgrund der Methoden **`wait()`** und **`join()`** mit einer Zeitangabe
  - **TERMINATED**: Der Thread ist beendet
- Der Status kann mittels **`getState()`** erfragt werden

Starten eines Threads

- Es kann nur ein Objekt der Klasse **`Thread`** als eigenständiger Thread gestartet werden
- Wurde das Interface **`Runnable`** implementiert, muss ein Objekt dieser Klasse zunächst in ein **`Thread`**-Objekt gewrappt werden:
```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
```
- Ruft ein Programmierer am **`Thread`**-Objekt die Methode **`run()`** auf, werden die Anweisungen ausgeführt, aber es wird kein neuer eigenständiger Thread gestartet
- Nur die Methode **`start()`** der Klasse **`Thread`** erzeugt einen neuen Thread und ruft dann die Methode **`run()`** auf
- Beispiel: **`thread1`**

Starten eines Threads

- Die Methode **`start()`** kann an einem **`Thread`**-Objekt immer nur einmal aufgerufen werden
- Soll ein Runnable mehrmals nacheinander gestartet werden oder aber mehrere **`Runnable`**-Objekte in einem Thread nacheinander abgearbeitet werden ist eine Verwendung eines **`Thread`**-Objektes nicht möglich
- Seit Java5 gibt es für derartige Probleme die Klasse **`Executor`**

Starten eines Threads

- Man erhält einen Executor, indem man die Klasse **`Executors`** beauftragt einen zu erzeugen:
```
Executor executor =
Executors.newCachedThreadPool();
```
- **`Runnable`**-Objekte können dann mittels der Methode **`execute()`** ausgeführt werden:
```
executor.execute (new MyRunnable1 ());
executor.execute (new MyRunnable2 ());
executor.execute (new MyRunnable1 ());
```
- Soll ein Executor keine weiteren Threads mehr annehmen, kann dieser geschlossen werden:
```
executor.shutdown();
```
- Nach dem Schließen eines Executors werden die bereits vorhandenen Threads weiter verarbeitet

Starten eines Threads

- Die Klasse **`Executors`** kann verschiedene Strategien zur Ausführung von Threads zur Verfügung stellen:
  - **`ScheduledThreadPool`** (Zeitgesteuerte Ausführung)
  - **`CachedThreadPool`** (Verwertet ggf. vorhandene Threads)
  - **`FixedThreadPool`** (Verwertet eine feste Anzahl von Threads wieder)
  - **`SingleThreadExecutor`** (Arbeitet alle **`Runnable`**–Objekte nacheinander ab)
  - **`NewThreadExecutor`** (Startet jedes **`Runnable`** in einem neuen Thread)

Beenden eines Threads

- Generell wird ein Programm/Prozess erst dann beendet, wenn der letzte Thread beendet ist
- Soll ein Thread beendet werden, geschah dies früher mit der Methode **`stop()`**
  - Diese Methode ist **`deprecated`**
  - Es ist nicht klar, wann ein Thread mittels **`stop()`** unterbrochen wird
  - Daher nicht mehr verwenden
- Besser ist es den Status eines Threads zu ändern

Beenden eines Threads

- Einem Thread kann mit der Methode **`interrupt()`** ein Signal gegeben werden, dass er abbrechen soll
- Innerhalb des Threads kann nun mit der Methode **`isInterrupted()`** dieser Status abgefragt werden
- Ist als Threadstatus der Abbruch eingetragen, kann nun innerhalb des Threads der Abbruch vorgenommen werden
- In der Regel bedeutet dies, das eine Schleife beendet wird, die in der **`run()`** Methode implementiert ist
- Auf diese Weise kann ein Thread selbst bestimmen, wann er abbricht und kann ggf. noch bestimmte Aufgaben/Aufräumarbeiten erledigen, bevor er anhält
- Beispiel: **`thread2`**

Threads schlafen schicken

- Ein Thread kann kurzzeitig in einen Ruhezustand versetzt werden
- Dazu gibt es zwei Methoden:
  - Die überladende Klassenmethode **`Thread.sleep()`**
    - Es handelt sich hierbei um eine Klassenmethode, da **`sleep()`** nicht von außen an ein **`Thread`**-Objekt geschickt werden soll
    - Nur so kann verhindert werden, einen fremden Thread, über dessen Referenz man verfügt, ein paar Sekunden lang schlafen zu legen
  - Die Instanzmethode **`sleep()`** an einem **`TimeUnit`**-Objekt
    - Auch diese Methode bezieht sich immer auf den ausführenden Thread
    - Der Vorteil gegenüber **`sleep()`** ist die einfachere Verwendung von Zeiteinheiten

Threads schlafen schicken

- Die **`sleep()`** Methoden werfen unter Umständen eine **`InterruptedException`**
- Dies ist immer dann der Fall, wenn bei einem Thread der Status mittels **`interrupted()`** geändert wird, während der Thread schläft
- Da die **`InterruptedException`** keine **`RuntimeException`** ist, muss der Aufruf von **`sleep()`** in einem **`try/catch`**-Block stehen
- Eine entsprechende **`throws`**-Klausel im Methodenkopf ist aufgrund der Vererbung der Methode **`run()`** nicht möglich
- Beispiel: **`thread3`**

Threads pausieren lassen

- Neben **`sleep()`** gibt es eine weitere Methode, um kooperative Threads zu programmieren
- Die Methode **`yield()`**
- Sie funktioniert etwas anders als **`sleep()`**
- Mittels **`yield()`** kehrt man nicht nach Ablauf der genannten Millisekunden zum Thread zurück
- **`yield()`** ordnet den Thread bezüglich seiner Priorität wieder in die Thread-Warteschlange des Systems ein
- Einfach ausgedrückt, sagt **`yield()`** der Thread-Verwaltung: „Ich will jetzt nicht mehr, ich mache weiter, wenn ich das nächste Mal dran bin.“

Warten auf einen Thread

- Werden Aufgaben auf mehrere Threads verteilt, kommt ein Zeitpunkt, an dem die Ergebnisse eingesammelt werden sollen
- Die Resultate können erst dann zusammengebracht werden, wenn alle Threads mit ihrer Ausführung fertig sind
- Mit der Instanzmethode **`join()`** kann auf einen anderen Thread gewartet werden
  - **`final void join() throws InterruptedException`**
    Der aktuell ausgeführte Thread wartet auf den Thread, für den die Methode aufgerufen wird, bis dieser beendet ist
  - **`final void join(long millis) throws InterruptedException`**
    Wie **`join()`**, doch wartet diese Variante höchstens **`millis`** Millisekunden. Wurde der Thread bis dahin nicht vollständig beendet, fährt das Programm fort
  - **`final void join (long millis, int nanos) throws InterruptedException`**
    Wie **`join(long)`** jedoch mit potenziell genauerer Angabe der maximalen Wartezeit
- Beispiel: **`thread4`**

Threads unterschiedlicher Priorität

- Jeder Thread verfügt über eine Priorität, die aussagt, wie viel Rechenzeit ein Thread relativ zu anderen Threads erhält
- Die Priorität ist eine Zahl zwischen **`Thread.MIN_PRIORITY`** (1) und **`Thread.MAX_PRIORITY`** (10)
- Durch den Wert kann der Scheduler erkennen, welchem Thread er den Vorzug geben soll, wenn mehrere Threads auf Rechenzeit warten
- Bei seiner Initialisierung bekommt jeder Thread die Priorität des erzeugenden Threads, Normalerweise die Priorität **`Thread.NORM_PRIORITY`** (5)
- Das Betriebssystem (oder die JVM) nimmt die Threads anteilig nach der Priorität aus der Warteschlange heraus
- Die Priorität kann durch Aufruf von **`setPriority()`** geändert und mit **`getPriority()`** abgefragt werden
- Java macht aber nur sehr schwache Aussagen über die Bedeutung und Auswirkung von Thread-Prioritäten
- Beispiel: **`thread5`**

Dämonen

- Ein Server horcht auf eingehende Aufträge oft in einer Endlosschleife und führt die gewünschte Aufgabe aus
- Enthält ein gestarteter Thread eine Endlosschleife wird dieser grundsätzlich nie beendet
- Der Thread würde also immer weiter laufen, auch wenn die Hauptapplikation beendet ist
- Dies ist nicht immer beabsichtigt, da z.B. die Server-Funktionalität nach Beenden der Applikation nicht mehr gefragt ist
- Auch der Endlos-Thread sollte beendet werden
- Dazu kann ein Thread als Dämon gekennzeichnet werden
  - Dämon bezeichnet eigentlich ein Geisterwesen, einen Schutzgeist, Mischwesen (Chimäre)
- Standardmäßig ist ein Thread kein Dämon

Dämonen

- Ein Dämon ist im Hintergrund mit einer Aufgabe beschäftigt
- Wenn das Hauptprogramm beendet ist und die Laufzeitumgebung erkennt, dass kein normaler Thread läuft, sondern nur Dämonen, dann werden diese ebenfalls beendet
- Man muss sich um das Ende eines Dämonthreads nicht kümmern
- Einen Thread kann mit der Methode **`setDaemon()`** als Dämon gekennzeichnet werden
- Der Aufruf der Methode ist nur vor dem Starten des Threads erlaubt
- Nach dem Start kann der Status nicht geändert werden
- Beispiel: **`thread6`**

Fehlerbehandlung bei Threads

- Treten innerhalb eines Threads Exceptions auf, die nicht innerhalb des Threads behandelt werden, können diese nicht ohne weiteres abgefangen werden (wo will man den Fehler des anderen Threads abfangen?)
- Daher kann an einen Thread ein **`UncaughtExceptionHandler`** angehängt werden
- Dieser wird immer benachrichtigt, wenn ein Thread wegen einer nicht behandelten Exception beendet wird
- **`UncaughtExceptionHandler`** ist ein Interface, welches innerhalb der Klasse **`Thread`** definiert wird
- Diese Vorgehensweise wurde gewählt, da das Interface ausschließlich im Zusammenhang mit der Klasse **`Thread`** verwendet wird
- In dem Interface wird nur die Methode **`void uncaughtException(Thread t, Throwable e)`** definiert, die aufgerufen wird, insofern eine nicht gefangene Exception innerhalb eines Threads geworfen wird

Fehlerbehandlung bei Threads

- Eine Implementierung des Interfaces lässt sich entweder einem individuellen Thread oder allen Threads anhängen
- Im Falle eines Abbruchs durch eine unbehandelte Exception ruft die JMV die Methode **`uncaughtException()`** auf
- Auf diese Weise kann innerhalb der Methode noch auf den Fehler reagiert werden, den die JVM über das **`Throwable e`** übergibt

Fehlerbehandlung bei Threads

- Methoden zur Verwaltung der **`UncaughtExceptionHandler`**
  - **`void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)`**
    Setze den **`UncaughtExceptionHandler`** für den Thread.
  - **`Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()`**
    Liefert den aktuellen **`UncaughtExceptionHandler`**.
  - **`static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)`**
    Setze den **`UncaughtExceptionHandler`** für alle Threads.
  - **`static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()`**
    Liefert den zugewiesenen **`UncaughtExceptionHandler`** aller Threads.
- Wenn ein mit **`setUncaughtExceptionHandler()`** lokal gesetzter **`UncaughtExceptionHandler`** gesetzt ist, wird der über die Methode **`setDefaultUncaughtExceptionHandler()`** gesetzte DefaultHandler nicht verwendet
- Beispiel: **`thread7`**

`Timer`

- Zeitgesteuerte Aktionen können mittels eines **`Timer`**-Objektes durchgeführt werden
- Um eine Aktion zeitgesteuert durchführen zu können muss zuerst eine Klasse implementiert werden, die von der Abstrakten Klasse **`TimerTask`** abstammt
- Dazu muss in der Unterklasse (ähnlich wie bei einem Thread) die Methode **`run()`** implementiert werden
- Des Weiteren muss ein **`Timer`**-Objekt erzeugt werden
- Das **`Timer`**-Objekt kann dann mit der Methode **`schedule`** den Task zeitgesteuert ausführen
- Dabei kann ein Task einmalig oder aber wiederholt ausgeführt werden
- Ein **`Timer`**-Objekt wird mit der Methode **`cancel()`** beendet
- Beispiel: **`timer1`**

ShutdownHook

- Ein Javaprogramm kann normal zu Ende gehen oder mit „Gewalt“ beendet werden, dazu wird z.B. die Tastenkombination Strg+C auf der Kommandozeile eingegeben
- Dabei wird ein Signal an die JVM geschickt und das Programm wird beendet
- Will man noch vor der Beendigung des Programms z.B. Aufräumarbeiten erledigen, kann man einen Thread einhängen, der die Aufgabe übernimmt

```
Thread t = …;
Runtime.getRuntime().addShutdownHook(t);
```

- Der Thread **`t`** wird dann (fast) immer vor Beendigung eines Programms durchgeführt, insofern es normal beendet wurde, oder das Signal durch die Tastenkombination Strg+C geschickt wurde (geht nicht in Eclipse oder mit dem Windows Taskmanager)
- Beispiel: **`shutdownhook1`**

Gemeinsamer Zugriff

- Ein Thread besitzt seinen eigenen Variablenraum
- Threads können aber auch Ressourcen gemeinsam nutzen, z.B. Klassenvariablen
- Beispiel: **`thread8`**
- In diesem Fall können verschiedene Exemplare einer Thread Klasse, Daten austauschen, indem sie Informationen ablegen oder entnehmen
- Threads können aber auch an einer zentralen Stelle eine Datenstruktur erfragen und dort Informationen entnehmen oder Zugriff auf gemeinsame Objekte über eine Referenz bekommen
- Es gibt also viele Möglichkeiten, wie Threads – und damit potenziell parallel ablaufende Aktivitäten – Daten austauschen können
- Hierdurch können sich verschiedene Probleme ergeben

Gemeinsamer Zugriff

- Dass Threads ihre eigenen Daten verwalten ist kein Problem, dieser Bereich ist geschützt
- Wenn mehrere Threads gemeinsame Daten nur lesen, ist das unbedenklich
- Schreiboperationen sind jedoch kritisch
- Beispiel:
  - Mehrere Nutzer teilen sich einen Drucker
  - Die Ausdrucke werden nicht in Einheit pro Nutzer gebündelt
  - Dadurch werden Seiten, Zeilen oder einzelne Zeichen aus verschiedenen Druckaufträgen bunt gemischt ausgedruckt

Gemeinsamer Zugriff

- Die Probleme entstehen durch die Umschaltung der einzelnen Threads
- Der Scheduler unterbricht zu einem unbekannten Zeitpunkt die Abarbeitung eines Threads und lässt den nächsten arbeiten
- Wenn der erste Thread gerade kritische Programmzeilen abarbeitet, die zusammengehören, und der zweite Thread beginnt, parallel auf diesen Daten zu arbeiten, kommt es ggf. zu Problemen

Zugriff auf ein Bankkonto

Beispiel:

- Gegeben seien Thread1 und Thread2 die ein Bankkonto manipulieren wollen, der Kontostand beträgt 100 Euro
- Thread1 will 20 Euro abbuchen, Thread2 will 50 Euro gutschreiben
  - Thread1 liest den Kontostand (100 Euro)
  - Thread1 berechnet den neuen Wert (80 Euro)
  - Thread1 will den Wert ins Konto schreiben, wird jedoch vom Scheduler unterbrochen und Thread2 ist dran
  - Thread2 liest den Kontostand (100 Euro)
  - Thread2 erhöht den Kontostand (150 Euro)
  - Thread2 schreibt den Kontostand (150 Euro)
  - Thread1 ist wieder dran und schreibt den Kontostand (80 Euro)
- Es ist notwendig Ressourcen für den exklusiven Zugriff zu sperren oder kritische Programmzeilen unterbrechungsfrei durchzuführen
- Beispiel: **`thread9`**

Kritisch / Nichtkritisch

- Zusammenhängende Programmblöcke, die nicht unterbrochen werden dürfen und besonders geschützt werden müssen, nennen sich kritische Abschnitte
- Wenn immer nur ein Thread den Programmteil abarbeitet, dann nennt man dies gegenseitigen Ausschluss oder atomar
- Wenn mehrere Threads auf das gleiche Programmstück zugreifen muss das nicht zwangsläufig zu einem Problem führen
- Immutable Objekte müssen nicht thread-sicher sein, denn es gibt keine Schreibzugriffe und bei Lesezugriffen kann nichts schief gehen
- Das gleiche gilt für Methoden, die keine Objekteigenschaften verändern (z.B. lesende Zugriffe)
- Da jeder Thread seine Thread-eigenen Variablen besitzt können lokale Variablen, auch Parametervariablen, beliebig gelesen und geschrieben werden

i++

- Wichtig ist zu wissen, welche Anweisungen atomar sind
- Die Anweisung **i++** sieht auf den ersten Blick atomar aus
- Sei folgende Klasse gegeben:

```
public class IPlusPlus {
    static int i = 0;
    public static void main(String args[]) {
        i++;
    }
}
```

- Sieht der Bytecode folgendermassen aus:

```
public static void main(java.lang.String[]);
   Code:
   0:   getstatic       #10; //Field i:I
   3:   iconst_1
   4:   iadd
   5:   putstatic       #10; //Field i:I
   8:   return
```

Kritische Abschnitte schützen

- Soll die Laufzeitumgebung nur einen Thread in einen Block lassen, benötigt man einen Monitor
- Ein Monitor ist ein Objekt, welches den einzelnen Threads bekannt sein sollte
- Tritt ein Thread in einen kritischen Abschnitt ein, kann die JVM den Monitor als belegt kennzeichnen
- Kommt ein zweiter Thread zu einem abgeschlossenen kritischen Bereich, muss er warten und wird erst hineingelassen, wenn die Markierung gelöscht ist
- Erst wenn der Thread den kritischen Bereich beendet hat, gibt die JVM den Monitor wieder frei, und ein anderer Thread kann den kritischen Bereich betreten
- Die Überwachung wird von der JVM übernommen

Rückblick

- Threads
- Erzeugen, Starten und Stoppen
- Fehler abfangen
- Gemeinsamer Zugriff auf Ressourcen
- Kritische Blöcke

Ausblick

- Synchronisieren von Threads