FORTGESCHRITTENE PROGRAMMIERKONZEPTE

Kurzbeschreibung

In dieser Kurseinheit wird mit der Ausnahmebehandlung und den *Threads* auf zwei wichtige fortgeschrittene Konzepte der Programmierung eingegangen und deren Umsetzung in Java wird aufgezeigt.

Schlüsselwörter

Ausnahme, Fehlercode, Ausnahmebehandlung, Klasse Exception, try-catch-Konstrukt, Parallelität, *Thread*, Klasse Thread, Synchronisation, kritischer Bereich, Sperrvariable

Lernziele

1. Die Vorteile eines Ausnahmebehandlungskonzepts gegenüber Fehlercodes werden verstanden und die hierfür bereitgestellten Sprachelemente können bei der Erstellung von Programmen gezielt eingesetzt werden.
2. Das Konzept der *Threads* sowie deren Synchronisation werden durchdrungen und die Umsetzung dieses Konzepts in einfache Programme wird beherrscht.

Hauptquellen

- Hanspeter Mössenböck: Sprechen Sie Java? – Eine Einführung in das systematische Programmieren, dpunkt.verlag 2003.

Inhaltsverzeichnis

1 AUSNAHMEBEHANDLUNG 2
1.1 Fehlercodes 2
1.2 Trennung der Fehlerbehandlung 4
1.3 Klasse Exception 5
1.4 Auslösen und Behandeln einer Ausnahme 7
2 THREADS 9
2.1 Quasiparallelität 9
2.2 Synchronisation von *Threads* 12
2.3 Deadlock 15
VERZEICHNISSE 18
Abkürzungen und Glossar 18
Index 18
Informationen und Interaktionen 18
Literatur 19

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- AUSNAHMEBEHANDLUNG
  - Grundlegende Konzepte, Arten von Ausnahmen, Klasse Exception, Auslösen und Behandeln einer Ausnahme
- THREADS
  - Parallelität, Klasse Thread, Erzeugung, Synchronisation, synchronized-Anweisung, kritischer Bereich, Deadlock, Monitor

Information 1: FORTGESCHRITTENE PROGRAMMIERKONZEPTE

1 AUSNAHMEBEHANDLUNG

Ein nicht zu unterschätzender Teil eines Programms besteht darin, Fehlersituationen zu behandeln, die z.B. aufgrund eines kurzfristig entstandenen Ressourcenmangels oder einer fehlerhaften Benutzereingabe im Programmverlauf aufgetreten sind [Mö03].

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Im Ablauf eines Programms können vielfältige Fehlersituationen auftreten, die vom Programm zu behandeln sind
- Zwei Möglichkeiten des Umgangs mit Fehlersituationen
  1. Einführung von Fehlercodes, die von Methoden im Falle einer festgestellten Fehlersituation zurückgeliefert werden
  2. Konzepte zur Ausnahmebehandlung (Exception Handling), die von modernen Programmiersprachen angeboten werden

Information 2: AUSNAHMEBEHANDLUNG - Problemstellung und Lösungsansätze

Grundsätzlich bestehen die zwei in Information 2 genannten Möglichkeiten zum Umgang mit den verschiedenen Fehlerarten.

Bevor das Konzept der Ausnahmebehandlung am Beispiel der Sprache Java vorgestellt wird, soll zunächst eine konventionelle Technik – die Einführung von Fehlercodes – behandelt werden. Aus den Nachteilen der Fehlercodes wird die dann ausführlicher beschriebene Ausnahmebehandlung motiviert.

1.1 Fehlercodes

Die Idee der Fehlercodes besteht darin, dass Methoden eventuell festgestellte Fehler an die aufrufende Methode in Form eines Rückgabewertes melden.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Fehlercodes sind Funktionswerte, die von einer Methode zurück geliefert werden
- Beispiel eines Fehlercodes
  - Methode enterCourse() liefert als Ergebnis
    - 0 (ok)
    - 1 (overflow)
    - 2 (numberExists)
    - 3 (...)
- Problem: Fehlerbehandlung verschlechtert die Lesbarkeit des Programms bei mehreren Methodenaufrufen erheblich

```
result = f();
if (result == ok) {
  result = g();
    if (result == ok) {
      result = h();
      if (result == ok) {
        ...
      } else ... // error handling
    } else ... // error handling
  } else ... // error handling
}
```

Wie würde das Programm ohne Abfrage von Fehlercodes lauten?

____

Interaktion 1: Einführung von Fehlercodes

Es kann beispielsweise für die Methode enterCourse() folgender Fehlercode vereinbart werden:

- ok: es ist kein Fehler in der Methode aufgetreten.
- overflow: der Kurs konnte nicht in den Katalog aufgenommen werden, weil die maximale Kapazitätsauslastung des Kurskatalogs erreicht ist.
- numberExists: die Kursnummer ist bereits vorhanden, weshalb eine Aufnahme des Kurses in den Katalog nicht möglich ist.

Die Fehlercode-Technik weist den großen Nachteil auf, dass der eigentliche (fehlerfreie) Ablauf des Programms durch die nach jedem Methodenaufruf erforderliche Abprüfung des Fehlercodes nicht mehr klar erkennbar ist. Das Beispielprogramm in Interaktion 1, das die Abfolge von drei Methodenaufrufen skizziert, macht dieses Problem deutlich.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Trennung der Fehlerbehandlung vom Normalablauf
  - Fehlermeldung nicht über Rückgabewerte zu realisieren, die eine rufende Methode zu einer Fehlercode-Überprüfung zwingt
- Flexibilität der Fehlerbehandlung
  - eine beliebige Methode in der Aufrufkette soll auf eine festgestellte Fehlersituation reagieren können
- Garantierte Fehlerbehandlung
  - jeder mögliche Fehler wird garantiert von irgendeiner Methode in der Aufrufkette bearbeitet

Information 3: Anforderungen an die Fehlerbehandlung

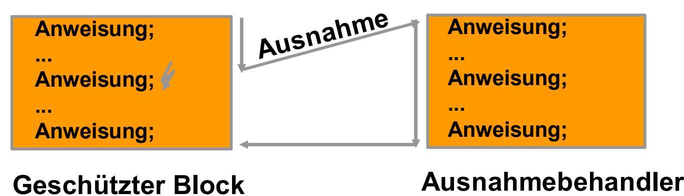
Aus den Nachteilen, die durch die Fehlercodes entstehen, resultiert die erste, in Information 3 genannte Anforderung der Trennung der Fehlerbehandlung vom Normalablauf, der die

eigentliche Algorithmusbeschreibung darstellt. Die zwei weiteren Anforderungen der Flexibilität und der garantierten Bearbeitung von festgestellten Fehlersituationen führen zum Konzept der Ausnahmebehandlung (*Exception Handling*).

1.2 Trennung der Fehlerbehandlung

Die notwendige Trennung der Fehlerbehandlung erfolgt durch die Einführung von so genannten Ausnahmebehandlern (*Exception Handler*).

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)



- Ein Block kann vor dem Auftreten von Fehlern geschützt werden
  - tritt ein Fehler auf, wird eine Ausnahme (exception) erzeugt
  - die Ausnahme wird durch einen Ausnahmebehandler bearbeitet
  - danach wird die Bearbeitung des Programms an der Stelle nach dem geschützten Block fortgesetzt

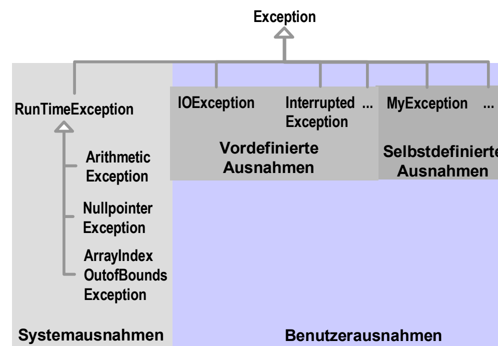
Information 4: Konzept der Ausnahmebehandlung

Ein Block, in dem eine Fehlersituation auftreten kann, wird durch ein geeignetes Sprachelement geschützt. Tritt eine Ausnahme tatsächlich auf, wird der entsprechende für diese Ausnahme zuständige Ausnahmebehandler aufgerufen. Nach Behandlung der Ausnahme setzt das Programm mit der ersten Anweisung hinter dem geschützten Block (*Protected Block*) fort. Den Programmfluss bei Auftreten einer Ausnahme im geschützten Block zeigt die Abbildung in Information 4.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Der Programmausschnitt zeigt die einfachste Form einer try-Anweisung
- Arten von Ausnahmen in Java
  - Systemausnahmen
    - ArithmeticException
    - NullPointerException
    - ArrayIndexOutofBounds Exception
  - Benutzerausnahmen
    - durch eine throw-Anweisung ausgelöst, um Fehler zu signalisieren
    - vordeklarierte (z.B. java.io.IOException) und vom Programmierer selbst definierte Ausnahmen (z.B. MyException)

```
try {
  ...        // protected block
} catch (Exception e) {
  ...        // exception handler
}
```



Information 5: try-Anweisung und Arten von Ausnahmen

In Java steht als Sprachelement zur Ausnahmebehandlung die try-Anweisung zur Verfügung, deren einfachste Ausprägung im Programmausschnitt in Information 5 gezeigt wird. Hierdurch lässt sich eine Ausnahme im geschützten Block "einfangen" (catch()) und die Bearbeitung dieser Ausnahme kann in einem nachfolgenden Ausnahmebehandler-Block durchgeführt werden.

Es werden in Java zwei Arten von Ausnahmen – die Systemausnahmen und die Benutzerausnahmen – unterschieden:

- Systemausnahmen entstehen aufgrund von illegalen, vom Programm ausgeführten Instruktionen, wie z.B. die Division durch 0 (ArithmeticException), der Zugriff auf ein Objekt über einen Null-*Pointer* (NullPointerException) oder ein außerhalb der Indexgrenzen liegender Array-Zugriff (ArrayIndexOutOfBoundsException).

- Benutzerausnahmen werden durch unzulässige Eingaben des Benutzers verursacht. Sie werden explizit durch die weiter unten behandelte throw-Anweisung ausgelöst. Neben den von Java vordefinierten Benutzerausnahmen, wie z.B. eine bei der Ein-/Ausgabe auftretende Ausnahme (java.io.IOException) kann der Programmierer eigene, d.h. selbst definierte Benutzerausnahmen einführen.

1.3 Klasse Exception

Hinter Ausnahmen verbirgt sich eine Klasse Exception, die als Datenanteil Informationen über den aufgetretenen Fehler beinhaltet und als Funktionsanteil verschiedene Methoden zum Zugriff auf diese Informationen bereitstellt. Die oben beschriebenen Ausprägungen von Ausnahmen entstehen dadurch, dass von der Klasse Exception Unterklassen abgeleitet werden. Die hieraus resultierende Klassenhierarchie zeigt die Abbildung in Information 5.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

```
class Exception {
  String toString();          // describes type of exception
  void printStackTrace();     // prints chain of called methods
  ...
}
```

- Basisklasse (Wurzel) der Vererbungshierarchie
  - Daten und Methoden werden von allen Exception-Unterklassen übernommen und ergänzt
- Zugriff über ein Exception-Objekt e
  - e.toString() beschreibt die Art des Fehlers und kann in Fehlermeldungen verwendet werden
  - e.printStackTrace() gibt die Methodenaufrufkette bis zur Methode des Java-Laufzeitsystems aus, durch die das Programm gestartet wurde

Information 6: Klasse Exception

Information 6 beschreibt zwei wichtige Methoden aus der von der Klasse Exception angebotenen Schnittstelle [MS+03].

Die Methode printStackTrace() liefert beispielsweise wichtige Informationen zum Inhalt des Laufzeitkellers, der zum Zeitpunkt bestand, als die Ausnahmesituation aufgetreten ist. Zu diesen Informationen gehört die Methodenaufrufkette, die in der Kurseinheit PROGRAMMIER-GRUNDLAGEN [C&M-PG] behandelt werden. Nähere Ausführungen zum Laufzeitkeller und der damit verbundenen Datenstruktur finden sich in der Kurseinheit OBJEKTORIENTIERTE PROGRAMMIERUNG [C&M-OP].

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

```
class CourseOverflowException extends Exception {   // inherits all attributes and
                                                    // methods from exception
  String overflowElement;                           // additional attribute
  CourseOverflowException (String shortName) {      // constructor
    overflowElement = shortName;
  }
}
```

- Definition einer eigenen Ausnahmeklasse
  - als Unterklasse von Exception
    - im Beispiel: class CourseOverflowException
- Ergänzung um zusätzliche Attribute und/oder Methoden, um den Fehler geeignet behandeln zu können

Information 7: Selbstdefinierte Ausnahmen

Es lassen sich durch Verfeinerung der Klasse Exception eigene Ausnahmeklassen definieren, wie das Beispiel in Information 7 verdeutlicht.

Die Ausnahme wird durch Erzeugung eines neuen CourseOverflowException-Objekts innerhalb einer throw-Anweisung ausgelöst. Bei der Erzeugung einer Ausnahme durch den Konstruktor CourseOverflowException() ist das Kursobjekt Course c, das den Überlauf verursacht hat, als Parameter zu übergeben.

1.4 Auslösen und Behandeln einer Ausnahme

Die throw-Anweisung ist dann auszuführen, wenn in der Methode addCourse(), die einen Kurs zum Kurskatalog hinzufügen soll, ein Überlauf des Katalogs festgestellt wurde.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Die Ausnahme wird durch die throw-Anweisung ausgelöst, wenn ein Kurs wegen Überlaufs nicht in den Katalog aufgenommen werden kann

```
public void addCourse(String shortName, int number) throws CourseOverflowException {
  if (numberEntries < entries.length)  // catalog not yet full?
    entries[numberEntries++] = new Course(shortName, number); // course added to catalog
  else  throw ______________________________; // generation of exception
```

- Eine eventuell auftretende Ausnahme wird durch das try-catch-Konstrukt bei Aufruf und Ausführung der zur Klasse CourseCatalog gehörenden Methode addCourse() gefangen und behandelt

```
try { cc.addCourse("Info1", 1001);
} catch ( ____________________________ ) {
Out.println("CourseOverflowException caused by course " + ______________________);
// exception object and course title to be added
}
```

Interaktion 2: Auslösen und Behandlung einer Ausnahme

Im oberen Teil von Interaktion 2 ist die Methode, deren throw-Anweisung zu vervollständigen ist, angegeben.

Die Behandlung der von der Methode addCourse() ggf. erzeugten Ausnahme zeigt der untere Programmausschnitt in Interaktion 2. Der durch die try-Anweisung geschützte Bereich besteht hier nur aus einer Anweisung, dem Aufruf der Methode addCourse() zu einem Objekt cc der Klasse CourseCatalog. Sollte die (selbst definierte) Ausnahme CourseOverflowException auftreten, wird das mittels der obigen throw-Anweisung erzeugte Ausnahmeobjekt "eingefangen" und in diesem Fall durch eine Ausgabe behandelt. Die Ausnahmebehandlung ist geeignet zu ergänzen.

- try-Anweisungen können über mehrere Methodendeklarationen hinweg verschachtelt sein
  - es werden bei den innersten catch-Klausel startend alle umgebenden catch-Klauseln überprüft, bis der passende Typ der erzeugten Ausnahme gefunden wurde

```
void m1() {
  try { ...
    m2();
} catch (E1 e) {
  // 3 ...
} catch (E2 e) { ... }
  // 4 ...
}
```

```
void m2() {
  ...
  m3();
  ...
}
```

```
void m3() {
  ...
  try { ...
    if (...)
      throw new E1(); // 1
    else
      throw new E2(); // 2
  } catch (E2 e) { // 5...}
  // 6 ...
}
```

Welcher Programm-ablauf entsteht, wenn in m3()

- Ausnahme E1
  // 1 // ____________
- Ausnahme E2
  // 2 // ____________

ausgelöst wird?

Interaktion 3: Suche eines Ausnahmebehandlers

Das Programmbeispiel in Interaktion 3 macht deutlich, dass try-catch-Konstrukte verschachtelt sein können. Die Suche nach einer zu dem aufgetretenen Ausnahmetyp passenden catch-Klausel erfolgt über die Verschachtelungen hinweg und endet, sobald die erste catch-Klausel gefunden wurde. Nach Ausführung der dazu gehörigen Anweisungsfolge wird das Programm nach der try-Anweisung, zu der die catch-Klausel gehört, fortgesetzt.

Zu den beiden im Programm in Interaktion 3 geworfenen Ausnahmen E1 und E2 soll der jeweils zugehörige Ausnahmebehandler gesucht und der resultierende Programmablauf soll beschrieben werden.

- Das try-catch-Konstrukt kann durch eine finally-Klausel abgeschlossen werden
  - die Klausel wird immer (auch wenn gar keine Ausnahme aufgetreten ist) ausgeführt
  - dient dazu, die ggf. begonnenen Arbeiten im try-Block abzuschließen

```
try {
  ...
  ...
} catch (E1 e) {
  ...
} catch (E2 e) {
  ...
} finally {
  ...
}
```

- Welcher Ablauf ergibt sich, wenn im try-Block eine Ausnahme vom Typ E3 auftritt?

1. ____________________
2. ____________________
3. ____________________

Interaktion 4: finally-Klausel

Interaktion 4 geht auf eine spezielle Klausel, die finally-Klausel ein, die unabhängig davon, ob eine Ausnahme auftritt oder nicht, die Ausführung des try-catch-Konstrukts abschließt.

Im ausnahmefreien (Normal-) Fall würde im angegebenen Beispiel der vollständige try-Block und danach die finally-Klausel ausgeführt werden.
Entsprechend würde bei Auftreten einer der beiden Ausnahmen E1 bzw. E2 nach der entsprechenden catch-Klausel die finally-Klausel ausgeführt werden.
Der dritte Fall, das Auftreten einer Ausnahme, zu der keine catch-Klausel im try-catch-Konstrukt vorgesehen ist, wird im Rahmen von Interaktion 4 behandelt.

Mit den vorgestellten Konzepten zur Ausnahmebehandlung können alle Anforderungen, die weiter oben an ein Konzept einer effizienten und übersichtlichen Fehlerbehandlung gestellt wurden, erfüllt werden.

2 THREADS

Ein weiteres fortgeschrittenes und sehr mächtiges Programmierkonzept sind die *Threads*, durch die Teile eines Programms parallel ausgeführt werden können [Mö03].

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Ein Thread ist ein Programmstück, das parallel zu anderen Programmstücken abläuft
  - Programmstücke sind Prozesse, die parallel ablaufen
- Beispiel
  - Thread 1: Entgegennahme der Benutzereingabe
  - Thread 2: Durchführung von Berechnungen
  - Thread 3: Visualisierung von Zwischenergebnissen
- Parallelität bewirkt, dass sich die Threads nicht gegenseitig blockieren

Information 8: THREADS - Einführung

Die parallele Ausführung der in Information 8 genannten drei Beispiel-*Threads* bewirkt, dass eine Benutzereingabe (Thread 1) sofort entgegen genommen wird und nicht erst gewartet werden muss, bis beispielsweise eine angefangene Berechnung (Thread 2) beendet wurde oder ein Visualisierungsvorgang (Thread 3) abgeschlossen wurde.

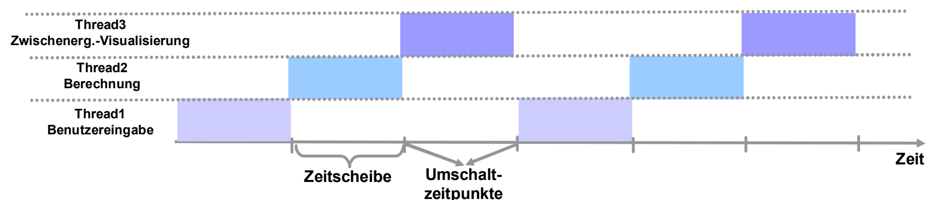
2.1 Quasiparallelität

Das *Thread*-Konzept, durch das eine parallele Ausführung von Programmteilen ermöglicht wird, lässt sich auch auf Ein-Prozessor-Rechnern realisieren, was zu der so genannten Quasiparallelität führt.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Echte Parallelität würde voraussetzen, dass mehrere Prozessoren zur Verfügung stehen

- Auf einem Rechner mit einem Prozessor werden die Threads quasiparallel ausgeführt
  - der Prozessor wird jedem der quasiparallel laufenden Threads nur eine gewisse (kurze) Zeitscheibe zugeordnet



Information 9: Quasiparallele Ausführung von *Threads*

Die quasiparallele Ausführung von *Threads* heißt, dass die beteiligten *Threads* den Prozessor nur jeweils eine gewisse Zeitscheibe zugewiesen bekommen.

Anhand der drei Beispiel-*Threads* wird die quasiparallele Verarbeitung in Information 9 veranschaulicht.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

```
public class Thread {
  public void start() {...]
  public static void sleep(int milliSeconds) {...}
  public void run() {...}
  ...
}
```

- Threads sind in Java Objekte der Klasse Thread
  - bietet u.a. Methoden zum Starten start() und Verzögern sleep()
  - wichtigste Methode ist run()
    - enthält den Code, den der Thread im Laufe seines Lebens ausführen soll

- Implementierung eines Threads, indem eine Unterklasse zur Klasse Thread gebildet wird und run() mit dem gewünschten Code des Threads überschrieben wird

Information 10: Klasse Thread

In Java wird eine Klasse Thread bereitgestellt, die den Ausgangspunkt der Implementierung von *Threads* bildet. Ein Ausschnitt aus der Klasse Thread und das Vorgehen zur *Thread*-Implementierung sind in Information 10 gegeben.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

```
class CharPrinter extends Thread {                    // thread CharPrinter is sub-class from Thread
  char signal;                                        // CharPrinter extends Thread by attribute signal

  public CharPrinter (char ch) { signal = ch;}        // constructor: gets a character ch from outside when called

  public void run() {                                 // run() method overwritten by CharPrint
    for (int i = 0; i <= 20; i++) {
      Out.print(signal);
      int delay = (int) (Math.random() * 100);        // delay is a random number between 0  and 99
      try { sleep(delay); } catch (Exception e) { return; }          // thread sleeps for delay seconds
    }                                                                // reason for try statement?
  }                                                                  // ___________________________
}
class ThreadCharPrinterProgram {
  public static void main (String[] arg) {
    CharPrinter thread1 = new CharPrinter('.'); CharPrinter thread2 = new CharPrinter('*');
    thread1.start(); thread2.start();                 // generate and start thread1 and thread2
    Out.print('+');
  }
}
```

Interaktion 5: Beispiel-Programm zu *Threads*

In Interaktion 5 ist ein Beispiel-Programm angegeben, in dem ein *Thread* CharPrinter implementiert ist, der in zufälliger zeitlicher Abfolge ein Zeichen ausgibt. Welches Zeichen vom *Thread* auszugeben ist, wird bei dessen Erzeugung festgelegt.

Die zufällige zeitliche Abfolge wird durch die von der Klasse Thread geerbte sleep()-Methode und einer mittels der random()-Methode zufällig gewählten Wartezeit erreicht. Die Methode sleep() kann jederzeit durch die Benutzereingabe Strg-C abgebrochen werden, weshalb eine entsprechende Ausnahmebehandlung vorzusehen ist.

In der main()-Methode werden dann zwei dieser *Threads* erzeugt, die jeweils ein unterschiedliches Zeichen (. bzw. *) ausgeben. Nach dem Start der beiden *Threads* gibt das main()-Programm vor dessen Beendigung ein Zeichen + aus.

- Das Java-Laufzeitsystem erzeugt für jedes Programm einen neuen Thread
  - run()-Methode startet die main()-Methode
- Jede run()-Methode hat ihre eigenen lokalen Variablen
- Threads sind in Java gewöhnliche Objekte, auf die andere Objekte gewöhnlich (über Zeiger auf dieses Thread-Objekt) zugreifen können
- Threads können ihrerseits auf andere Objekte zugreifen
  - Threads "teilen" sich diese Objekte, d.h. die Objektdaten werden nicht für jeden Thread kopiert
  - hierdurch ist die Kommunikation von Threads über die Objekt-Attribute möglich
  - erfordert allerdings ggf. einen synchronisierten Zugriff, um Inkonsistenten auszuschließen

Information 11: *Thread*-Konzept in Java

Mit Hilfe des *Thread*-Konzepts kann jetzt auch geklärt werden, was beim Start eines Programms abläuft. Die für das main()-Programm innerhalb der run()-Methode angelegten Variablen sind lokal, d.h. der nächste gestartete *Thread* greift nicht auf diese Variablen zu, sondern erhält einen eigenen Speicherbereich für seine lokalen Variablen.

Da *Threads* ganz normale Objekte in Java sind, können andere Objekte auf diese zugreifen und umgekehrt können *Threads* auch auf andere Objekte zugreifen. Beim Zugriff auf andere Objekte ist allerdings zu beachten, dass unterschiedliche *Threads* auf ein und dasselbe Objekt zugreifen. Solche (quasi-) parallelen Zugriffe können zu Konflikten führen und erfordern daher eine im Folgenden behandelte Synchronisation der Zugriffe.

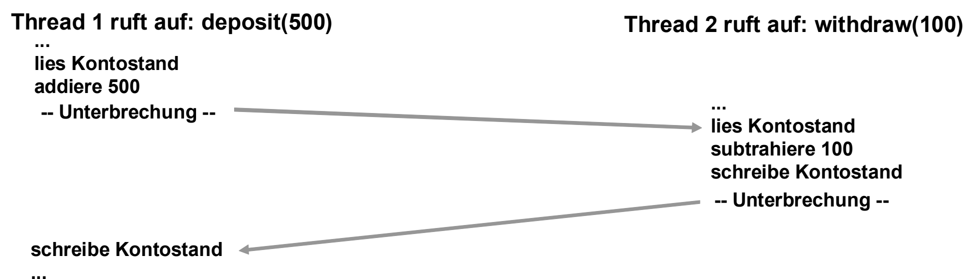
2.2 Synchronisation von *Threads*

An einem einfachen Beispiel soll zunächst erklärt werden, warum ein nicht abgestimmter Zugriff von mehreren *Threads* auf einen gemeinsamen Datenbestand zu unerwünschten Ergebnissen führen kann.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

```
class Account {
  int balance;
  void deposit (int x) { balance = balance + x; }
  void withdraw (int y) {balance = balance - y; }
}
```

- Gegeben sei eine Klasse Account zum Verwalten (Einzahlen und Abheben) von Bankkonten
  - zwei Threads, von denen der eine auf das Bankkonto einzahlt (deposit()) und der andere vom selben Bankkonto abhebt (withdraw())
  - Wie lautet der Kontostand (balance) nach Beendigung des angegebenen Ablaufs?



Interaktion 6: Motivation für die Synchronisation von *Threads*

In dem in Interaktion 6 angegebenen Beispiel arbeiten zwei *Threads* auf einem gemeinsamen Objekt, einem Bankkonto (class Account). Während Thread 1 500 Geldeinheiten auf das Konto einzahlt, hebt Thread 2 parallel dazu 100 Geldeinheiten ab. Ein solcher Vorgang kann in der Praxis durchaus auftreten, wenn z.B. ein Kontostand "gleichzeitig" durch eine Gutschrift erhöht und durch eine (z.B. am Geldautomaten getätigte) Abhebung verringert wird.

Die Synchronisationsproblematik entsteht dadurch, dass ein Java-Befehl – im Beispiel die beiden Methodenaufrufe deposit() und withdraw() – vom Compiler in mehrere Schritte zerlegt wird und grundsätzlich nach jedem dieser Schritte eine Unterbrechung erfolgen kann. Die Unterbrechung führt dann dazu, dass ein anderer *Thread* die Bearbeitung aufnimmt.

Der beispielhafte Ablauf verdeutlicht, dass auf diese Weise Fehler auftreten können. Im konkreten Ablauf wird die Einzahlung der 500 Geldeinheiten nicht berücksichtigt.

- Kritischer Bereich ist der Bereich im Programm, in dem auf gemeinsame Daten zugegriffen wird
  - im Beispiel ist das der Bereich, in dem deposit() und withdraw() auf balance zugreifen
- Der Fehler entsteht dadurch, dass ein Thread unterbrochen wird, während er sich im kritischen Bereich befindet
- Lösung des Problems: Verzögerung der Unterbrechung, bis der Thread seinen kritischen Bereich verlassen hat
  - Einführung von Sperrvariablen

Information 12: Kritischer Bereich

Das Beispiel macht deutlich, dass es Bereiche im Ablauf eines *Threads* gibt, die nicht unterbrochen werden sollten, da sonst die Gefahr von Fehlern aufgrund nicht abgestimmter Zugriffe auf gemeinsame Daten besteht. Solche Programmbereiche werden als kritische Bereiche (siehe Information 12) bezeichnet.

Die Verhinderung einer Unterbrechung lässt sich durch so genannte Sperrvariablen erreichen.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Sperrvariable ist ein beliebig erzeugtes dynamisches Objekt
  - vorzugsweise das Objekt, auf das nicht gemeinsam zugegriffen werden darf
- Im Beispiel ist balance kein Objekt, sondern eine int-Zahl, weshalb ein Objekt lock eingeführt wird:

```
class Account {
  int balance;
  Object lock = new Object();

  void deposit (int x) { synchronized lock {balance = balance + x;} }
  void withdraw (int y) { synchronized lock {balance = balance - y;} }
}
```

Information 13: Sperrvariable in Java

Wie das Beispiel in Information 13 zeigt, kann in Java der Schutz des kritischen Bereichs durch die synchronized-Anweisung erfolgen, in der die Sperrvariable angegeben wird. Die synchronized-Anweisung bewirkt, dass vor Ausführung der darin enthaltenen Anweisungen – also vor Betreten des kritischen Bereichs – überprüft wird, ob bereits ein anderer *Thread* diese Sperrvariable gesetzt hat, sich also in einem durch die Sperrvariable geschützten kritischen Bereich befindet. Erst wenn sichergestellt ist, dass sich kein anderer *Thread* in einem durch die Sperrvariable geschützten Bereich befindet, erhält der *Thread* die Erlaubnis, den kritischen Bereich zu betreten. Solange er diesen Bereich nicht verlässt, werden alle diejenigen *Threads* gesperrt, die an eine entsprechende synchronized-Anweisung gelangen, in der die Sperrvariable angegeben ist.

Durch die synchronized-Anweisung wird somit erreicht, dass die Schreibzugriffe auf gemeinsam genutzte Variablen atomar, also nicht zerteilbar, durchgeführt werden.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

```
class Account {
  int balance;

  synchronized void deposit (int x) { balance = balance + x; }
  synchronized void withdraw (int y) { balance = balance - y; }
  int getBalance() { return balance; }
}
```

- Java ermöglicht auch den Schutz ganzer Methoden durch die synchronized-Anweisung
- Eine Klasse, die Daten kapselt, auf die nicht gleichzeitig zugegriffen werden darf, wird auch als Monitor bezeichnet
- Erweiterung bzw. Aufweichung des Monitorkonzepts in Java
  - Methoden, die nur lesend auf die geschützten Daten zugreifen, müssen nicht als synchronized definiert werden (Beispiel: Methode getBalance())
  - Vorteil dieser Erweiterung:

Interaktion 7: Monitor

Anstelle der Einführung einer Variablen (im Beispiel lock), durch die mittels der synchronized-Anweisung eine Anweisungsfolge geschützt werden kann, ermöglicht Java auch den Schutz einer ganzen Methode. Das Beispiel in Interaktion 7 zeigt, wie das Problem des synchronisierten Zugriffs auf das Bankkonto mittels synchronized-Methoden gelöst werden kann. Werden in einer Klasse Methoden als synchronized deklariert, so wird während deren Ausführung sichergestellt, dass zu jedem Zeitpunkt immer nur höchstens eine dieser Methoden aktiv ist.

Eine Java-Klasse, in der alle Methoden als synchronized deklariert sind, wird in der Literatur auch als ein Monitor bezeichnet. In Java wird das Monitor-Konzept dahingehend erweitert bzw. aufgeweicht, dass Methoden, die nur lesend auf gemeinsame Daten zugreifen, nicht als synchronized definiert werden müssen. Das gilt allerdings nur für lesende Zugriffe, die atomar auf dem Rechner ausgeführt werden. Bei lesendem Zugriff auf Werte des Typs long oder double ist bei der heutigen Rechnergeneration daher ein Schutz mit synchronized erforderlich.

2.3 Deadlock

Beim Sperren von *Threads* muss darauf geachtet werden, dass nicht eine Situation eintritt, die als *Deadlock* oder Verklemmung bezeichnet wird.

www.cm-tm.uka.de/info1
Prof. Abeck & Info1-Team

- Ein Deadlock ist gegeben, wenn
  1. der sich im kritischen Bereich befindliche Thread erst dann weiterarbeiten kann, wenn eine bestimmte Bedingung erfüllt ist und
  2. diese Bedingung nur von einem der wartenden Threads erfüllt werden kann
- Beispiel
  - Das Konto darf nicht überzogen werden, d.h. withdraw(y) muss warten bis balance >= y
  - Die Bedingung kann nur durch Einzahlung auf das Konto erfüllt werden, aber deposit() ist gesperrt, weil sich withdraw() im kritischen Bereich befindet

Information 14: *Deadlock*

In einer *Deadlock*-Situation warten die *Threads* gegenseitig aufeinander: Der aktive *Thread* wartet darauf, dass einer oder mehrere der gesperrten *Threads* zur Erfüllung der Bedingung beiträgt und die gesperrten *Threads* auf den aktiven *Thread*, der aber nicht mehr weiterarbeiten kann.

Der *Deadlock* kann offensichtlich nur dadurch aufgehoben werden, dass der aktive *Thread* den kritischen Bereich freigibt und sich solange schlafen legt, bis sich die Bedingung erfüllt hat, die seine Weiterarbeit ermöglicht.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- wait() bewirkt, dass der rufende Thread
  1. den Monitor freigibt
  2. sich schlafen legt
- notify() bewirkt, dass
  1. der auf die Erfüllung der Bedingung wartende Thread geweckt
  2. weiterarbeitet, wenn die Bedingung erfüllt ist
  3. wieder schlafen gelegt wird, wenn die Bedingung (noch) nicht erfüllt ist
- Beispiel:

```
synchronized void deposit (int x)
  { balance = balance + x;
  notify();
}
```

```
synchronized void withdraw (int y)
  try {
    while (balance < y) wait();
    // balance <= y
  } catch (InterruptedException e) {
    return;
  }
  // balance >= y
  balance = balance - y;
}
```

Information 15: wait und notify

In Java stehen zur Vermeidung eines *Deadlocks* die beiden zur Klasse Object() gehörenden Methoden wait() und notify() zur Verfügung. Da die Klasse Account (wie alle anderen Klassen) von Object() abgeleitet ist, können diese Methoden innerhalb von deposit() und withdraw() in der in Information 15 angegebenen Form genutzt werden.

Mit den *Threads* und den Sprachelementen zu deren Synchronisation wurden die wichtigsten Sprachelemente, die zur Programmierung von parallelen Algorithmen zwingend erforderlich sind, einführend vorgestellt. Gleiches gilt für die Ausnahmebehandlung und die Sprachelemente, die im Zusammenhang mit der Klasse Exception stehen.

Die vorliegende Kurseinheit könnte um einige weitere fortgeschrittene Programmierkonzepte erweitert werden. Genannt seien exemplarisch Konzepte zur Realisierung von Programmen, die auf der Grundlage moderner Web-Technologien verteilt über das Internet arbeiten [AL03, CJ02, KR03].

VERZEICHNISSE

Abkürzungen und Glossar

Abkürzung oder Begriff	**Langbezeichnung und/oder Begriffserklärung**
Ausnahme Klasse Exception	Objekte, die einen Ausnahme- oder Fehlerzustand signalisieren. Eine Ausnahme wird mit der throw-Anweisung ausgelöst und kann mittels eines catch/finally-Konstrukts abgefangen und behandelt werden [MS+03].
Ausnahmebehandler	Programmabschnitt, der nach Auftreten einer in einem geschützten Block aufgetretenen Ausnahme ausgeführt wird. Englischer Begriff: *Exception Handler*
Deadlock	Situation, in der Threads gegenseitig aufeinander warten. Deutscher Begriff: Verklemmung
Fehlercode	Festlegung (Codierung) der in einer Methode ggf. auftretenden Fehler in Form von Rückgabewerten, die an die aufrufende Methode zu übergeben sind.
Geschützter Block	Programmblock, der durch einen Ausnahmebehandler geschützt ist. Englischer Begriff: *Protected Block*
Kritischer Bereich	Programmbereich im Ablauf eines *Threads*, die nicht unterbrochen werden sollten, da sonst die Gefahr von Fehlern aufgrund nicht abgestimmter Zugriffe auf gemeinsame Daten besteht.
Thread	Ein parallel zu anderen Programmstücken ausgeführtes Programmstück.

Index

Ausnahmebehandler 4
Deadlock 15
Fehlercode 2
geschützter Block 4
kritische Bereiche 14
Threads 9

Informationen und Interaktionen

Information 1: FORTGESCHRITTENE PROGRAMMIERKONZEPTE 2
Information 2: AUSNAHMEBEHANDLUNG - Problemstellung und Lösungsansätze 2
Information 3: Anforderungen an die Fehlerbehandlung 3
Information 4: Konzept der Ausnahmebehandlung 4
Information 5: try-Anweisung und Arten von Ausnahmen 5
Information 6: Klasse Exception 6
Information 7: Selbstdefinierte Ausnahmen 6
Information 8: THREADS - Einführung 9
Information 9: Quasiparallele Ausführung von *Threads* 10
Information 10: Klasse Thread 10
Information 11: *Thread*-Konzept in Java 12
Information 12: Kritischer Bereich 14

Information 13: Sperrvariable in Java 14
Information 14: *Deadlock* 16
Information 15: wait und notify 16

Interaktion 1: Einführung von Fehlercodes 3
Interaktion 2: Auslösen und Behandlung einer Ausnahme 7
Interaktion 3: Suche eines Ausnahmebehandlers 8
Interaktion 4: finally-Klausel 8
Interaktion 5: Beispiel-Programm zu *Threads* 11
Interaktion 6: Motivation für die Synchronisation von *Threads* 13
Interaktion 7: Monitor 15

Literatur

[AL+03] Sebastian Abeck, Peter C. Lockemann, Jochen Schiller, Jochen Seitz: Verteilte Informationssysteme – Integration von Datenübertragungstechnik und Datenbanktechnik, dpunkt.verlag, 2003.

[C&M-OP] Cooperation&Management: OBJEKTORIENTIERTE PROGRAMMIERUNG, Kursdokument zur Vorlesung "INFORMATIK I", http://www.cm-tm.uka.de/info1, Universität Karlsruhe (TH), C&M (Prof. Abeck).

[CJ02] David A. Chappell, Tyler Jewell: Java Web Services – Using Java in Service-Oriented Architectures, O'Reilly & Associates, 2002.

[KR03] Jim Kurose, Keith Ross: Computer Networking – A Top-Down Approach Featuring the Internet, Addison-Wesley, Pearson Education, 2003.

[Mö03] Hanspeter Mössenböck: Sprechen Sie Java? – Eine Einführung in das systematische Programmieren, dpunkt.verlag 2003.

[MS+03] Java – Programmierhandbuch und Referenz für die Java-2-Plattform, Standard Edition, dpunkt.verlag, 2003.