Paradigmen der Programmierung (2. Teil)

Prof. Dr. Erich Ehses

FH Köln
Abteilung Gummersbach

Wintersemester 2013/14

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Vorbemerkung

Die Aufgabe dieses Skripts besteht darin, Hintergrundinformation zu den Vorlesungsfolien zu liefern. Es fehlen allerdings manche Beispiele und vor allem Graphiken.

Lassen Sie sich durch die Stoffauswahl nicht irreführen: Es kann sein, dass hier das eine oder andere klausurrelevante Thema fehlt. Nur die Folien und Praktikumsaufgaben geben einen Überblick über die wichtigen Inhalte der Vorlesung.

Ich möchte Sie ermutigen, auch die angegebene Literatur zu Rate zu ziehen.

1.2 Überblick

Im nächsten Kapitel werden einige Neuerungen von Java 5 besprochen, die ich nicht alle in der Vorlesung diskutieren werde. Sie werden benötigt, um ein Programm lesbarer und einfacher auszudrücken – basta.

Anschließend geht es um das umfassendere Konzept der *Generischen Typen*. Die einfacheren Formen der Typparametrisierung kennen Sie ja bereits aus dem 2. Semester. Hier will vor allem die Probleme ansprechen, die im Zusammenhang mit Vererbung auftreten. Dabei stellt sich heraus, dass die in Java realisierte Lösung vielleicht nicht besonders glücklich ist.

Schließlich werden in den weiteren Kapiteln Fragen der Nebenläufigkeit diskutiert. Nebenläufigkeit weicht insofern von der prozeduralen Programmierung ab, als im Programm kein genauer Ablauf mehr erkennbar ist. Oberflächlich ist es daher mit der Objektorientierung verwandt. Da ihre Probleme immer wieder um die Fragen der Koordination des Programmablaufs und der Gültigkeit der Inhalte von Variablen kreisen, ist Nebenläufigkeit vom Charakter her aber viel technischer und viel mehr an der konkreten Ausführung orientiert als die Objektorientierung. Es ist eine Herausforderung, Objektorientierung in einer nebenläufigen Umgebung korrekt zu implementieren.

Anhand des in Scala realisierten Actor-Modells werde ich auch Konzepte vorstellen, die einen sicheren Umgang mit Nebenläufigkeit versprechen. Ebenso werden die höheren Mechanismen der Java-Bibliothek angesprochen.

Kapitel 2

Ergänzungen zu Java

Die Inhalte dieses Kapitels wurde zum Teil bereits in Algorithmen und Programmierung 2 vorgestellt. Sie sind hier nochmals der Vorständigkeit halber beschrieben. Dabei kommen auch gewisse Erweiterungen vor. In der Vorlesung werden einzelne Inhalte des Kapitels verwendet, ohne sie aber systematisch zu erläutern.

2.1 Annotationen

Definition:

> *Eine **Annotation** ist eine Aussage, die sich auf einen Typ, ein Datenelement einer Klasse oder auf eine Methode bezieht. Annotationen haben wie Kommentare keine unmittelbare Auswirkung auf die prozedurale Ausführung. Im Unterschied zu Kommentaren sind sie aber durch den Compiler, durch die Analyse des Classfiles oder sogar zur Laufzeit lesbar. Eine Annotation wird definiert durch ein **Annotationsinterface**. Die Anwendung einer Annotation erfolgt durch „@“ gefolgt von dem Annotationsnamen. Annotationen können durch das Interface festgelegte Parameter haben.*

Eine oft verwendete vordefinierte Annotation ist `@Override`. Sie steht bei einer Methode und sagt aus, dass diese eine Methode ihrer Oberklasse überschreibt. Diese Annotation wird vom Compiler ausgewertet und führt zu einer Fehlermeldung, wenn dies nicht richtig ist. Damit kann man nicht nur die Lesbarkeit sondern auch die Programmsicherheit erhöhen.

Eine andere häufig verwendete Annotation ist `@SuppressWarnings`. Diese Annotation kann bei Klassen, Feldern und Methoden stehen. Sie unterdrückt Compilerwarnungen im Zusammenhang mit diesen Elementen. Mit der Annotation muss eine Fehlerursache in Form eines Strings angegeben werden. Sollen mehrere Warnungen unterdrückt werden, werden ihre Namen in geschweiften Klammern gelistet.

Beispiel:

```
@SuppressWarnings(value = {"unchecked"})
class Unsinn extends Oberklasse {
     @Override
     public void method() {}
}
```

Die Angabe von `value =` in obiger Annotation kann entfallen. Ebenso die geschweiften Klammern. Sie stehen dafür, dass hier mehrere Strings angegeben werden können.

Der interessantere Teil ist die eigene Definition einer Annotation. Sie lässt sich am einfachsten an einem Beispiel erläutern.

```
import java.lang.annotation.*;

@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})

@interface MyAnnotation {
    String name();
    int alter() default 0;
    String[] freunde default {}
}
```

Ein Annotationsinterface wird durch das Wort `@interface` eingeleitet. Anschließend folgen Methodenköpfe. Diese haben keine Parameter, sondern nur Rückgabetypen. Erlaubt sind die vordefinierten skalaren Typen, die Typen `String`, `Class`, Enum-Typen (s.u.) und Arrays von den vorgenannten Typen.

Nach der Methodendeklaration kann optional das Schlüsselwort `default` mit der Angabe eines Defaultwertes stehen. Die Angabe von Defaults bewirkt, dass man in der Anwendung der Annotation das entsprechende Element weglassen kann. Dann wird automatisch der Defaultwert eingesetzt.

Mögliche Formen der Annotation:

```
@MyAnnotation{name = "Hans", alter = 18,
              freunde = {"Karl", "Karin"})
@MyAnnotation{name = "Hans", freunde = "Karin")
@MyAnnotation{name = "Hans")
```

Wenn eine Annotation nur ein Element hat, benennt man dieses in der Regel mit `value`. In der Anwendung kann `value =` auch weggelassen werden.

Die Annotationsdeklaration ist selbst durch Annotationen erläutert.

Die Annotation `@Target` gibt an, vor welchen Sprachelementen die Annotation stehen darf. Untern anderem ist erlaubt: `ElementType.TYPE`, `ElementType.FIELD` und `ElementType.METHOD`.

Die Annotation `@RetentionPolicy` bestimmt, wielange die Information abfragbar ist. Erlaubt sind die Angaben `RetentionPolicy.SOURCE` für die Verfügbarkeit im Compiler, `RetentionPolicy.CLASS` für das Speichern im Classfile und `RetentionPolicy.RUNTIME` für die Abfragbarkeit zur Laufzeit.

Die beiden anderen Annotationen werden seltener verwendet. `@Documented` bewirkt, dass die Annotation in Javadoc übernommen wird. Das ist nur dann nötig und sinnvoll, wenn die Annotation Auswirkungen auf die Verwendung des annotierten Elements hat. `@Inherited` bewirkt, dass die Annotation von einer annotierten Oberklasse automatisch an die abgeleiteten Klassen vererbt wird, sofern sie dort nicht überschrieben wird.

Im Nebenläufigkeitskapitel werden Annotationen verwendet, um die Eigenschaften von Klassen und Feldern bzgl. ihres Verhaltens bei Nebenläufigkeit zu beschreiben.

Es versteht sich von selbst, dass vom Programmierer definierte Annotationen nur dann eine Auswirkung haben, wenn sie entweder durch ein besonderes Compilerplugin oder durch Analyse des Classfiles oder der Klasse abgefragt werden. Die im System oder manchmal in einem Framework definierten Annotationen werden meist durch den Code des Frameworks ausgewertet.

2.2 Variable Anzahl von Parametern

Diese Spracherweiterung wird zwar nicht häufig verwendet, kann aber im Einzelfall sehr nützlich sein. Das typische Beispiel ist die Implementierung von `printf` in der Java-Bibliothek. Das aus C bekannte `printf` basiert darauf, dass man neben dem obligatorischen Formatstring so viele weitere Parameter angibt wie durch Platzhalter im Formatstring verlangt werden. `printf` ist somit eine Funktion mit einer im Voraus nicht festgelegten Anzahl von Argumenten.

Die variable Parameterzahl wird notiert, indem der Typ des letzten Parameters mit `T...` angegeben wird. `T` steht hierbei für den Namen des erwarteten Datentyps. Die Deklaration bewirkt, dass beim Methodenaufruf an dieser Stelle entweder eine beliebige Anzahl von Argumenten des Typs `T` oder ein Array von Typ `T[]` stehen darf. Innerhalb der Methode sind die Argumente als Elemente eines Arrays ansprechbar. Das folgende Beispiel zeigt die Definition und die Verwendung einer Methode, die einfach alle übergebenen Argumente zeilenweise ausgibt.

```
static void printAllArguments(Object ... args) {
    for (int i = 0; i < args.length; i++)
        System.out.println(args[i]);
 }

public static void main(String[] args) {
    printAllArguments("Hello", "Brave", "New", "World");
    String[] stringArray = {"Hello", "World"};
    printAllArguments(stringArray);
}
```

Der Kopf von `printf` sieht z.B. wie folgt aus:[1]

```
public PrintStream printf(String format, Object ... args)
```

`printf()` verwendet zur Formatierung die landesspezifischen Regeln. Sie dürfen sich also nicht wundern, dass Dezimalzahlen mit Komma ausgegeben werden. Für die Angabe des Zeilenumbruchs wird in Java `%n` gegenüber `\n` bevorzugt:

```
System.out.printf("%4.2f * %4.2f = %6.2f%n", a, b, a * b);
```

Ein anderes Beispiel für die variable Anzahl von Parametern ist die folgende Methode aus der Klasse `java.util.Arrays`:

[1] Beim Aufruf von `printf` können auch Ausdrücke mit Wertdatentypen, wie `int`, verwendet werden. Das ermöglicht der Compiler aber über das weiter unten besprochene *Autoboxing*, das eine automatische Erzeugung der passenden Objekte vornimmt.

```
public static <T> List<T> asList(T ... elements)
```

Diese Funktion ermöglicht es, ein vorhandenes Array als ein Objekt der Schnittstelle `List` anzusprechen, oder einfach aus einer Anzahl von Werten eine Liste zu definieren (der Typparameter `T` in dem Methodenkopf wird etwas später besprochen). Mögliche Anwendungen sind:

```
String[] array = {"a", "b", "c"};
List<String> liste1 = Arrays.asList(array);
List<String> liste2 = Arrays.asList("a", "b", "c");
```

Es bleibt anzumerken, dass auch Scala eine variable Parameterzahl zulässt, wenn der Datentyp von einem `*` gefolgt ist. Das Beispiel sollte alles klarmachen:

```
def printAllArguments(arguments: Any*): Unit =
  for (argument <- arguments) println(argument)
```

2.3 Autoboxing

Der Inhalt dieses Abschnitts sollte im wesentlichen aus AP2 bekannt sein.

Java hat die Eigenschaft, dass die Standarddatentypen keine Referenzdatentypen sind und dass sie sich nicht nach dem Paradigma der Objektorientierung verhalten. Sozusagen als Ausgleich bietet die Bibliothek Wrapper-Klassen an, deren einziger Zweck darin besteht, primitive Werte als Objekte zu verpacken, so dass sie sich z.B. in einem Behälterobjekt speichern lassen. Nun gibt es eine exakte Entsprechung von Wertdatentypen und Wrapper-Klassen: `int` entspricht `Integer`, `float` entspricht `Float` usw. Autoboxing bedeutet, dass der Compiler, da wo erforderlich, automatisch das Verpacken (Erzeugung eines Wrapper-Objekts) und das Entpacken vornimmt. Zum Beispiel können die beiden folgenden Zeilen:

```
Integer intWrapper = Integer.valueOf(4);
int zahl = intWrapper.intValue();
```

einfacher geschrieben werden als:

```
Integer intWrapper = 4;
int zahl = intWrapper;
```

Sinnvolle Beispiele als dieses finden sich in vielen Anwendungen, so auch in den Beispielen mit Behälterklassen. Autoboxing bringt zwar keine Laufzeitverbesserung mit sich, verbessert aber deutlich die Lesbarkeit.

2.4 Enum-Klasse

Dieser Abschnitt stellt über die schon aus AP2 bekannt Sachverhalte einige Erweiterungen des Enum-Konzepts dar.

Als letzte spätere Ergänzung von Java sollen hier Aufzählungen besprochen werden. Die Java enum-Möglichkeit schließt die aus C altbekannte Lösung der Enum-Anweisung ein, kann aber deutlich mehr, da ein Java-Aufzählungselement ein vollwertiges Objekt darstellt.

In der einfachsten Form definiert man mit `enum` typsichere Konstanten, d.h. man definiert einen Typnamen (Name der Enum-Klasse) und eine Anzahl von Konstanten. Diese Konstanten stellen die einzigen Instanzen der Klasse dar. Sie sind jeweils genau einmal vorhanden, so dass sie auch mit == verglichen werden können. Das Prinzip wird am folgenden einfachen Beispiel klar:

```
public enum Wochentag {
    MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,
    FREITAG, SAMSTAG, SONNTAG
}

...
Wochentag heute = Wochentag.MITWOCH;

...
if (heute == Wochentag.SONNTAG) ...
```

Die Klassenfunktion `values()` liefert eine Liste aller Werte, so dass sich leicht eine Schleife über alle Tage definieren lässt:

```
for (Wochentag t : Wochentag.values()) {
    ...
}
```

Mithilfe der Klasse `EnumSet`, lässt sich auch ein Bereich angeben:

```
for (Wochentag t : EnumSet.range(
    Wochentag.MONTAG, Wochentag.FREITAG) )
{
    ...
}
```

Die Methode `toString()` gibt den Namen der Konstanten lesbar aus.

Halten wir fest:

- Ein Enum definiert einen Typ.
- Ein Enum erzeugt eine festgelegte Anzahl von Objekten. Diese Objekte werden über die Enum-Konstanten angesprochen.
- Alle Enum Objekte verfügen über Methoden, die von der Oberklasse `Enum` geerbt werden oder automatisch vom Compiler generiert werden.
- Enum-Kassen dürfen auch eigene Methoden definieren.

Enum-Objekte werden einmalig durch das Enum-Konstrukt selbst erzeugt. Sie können wie andere Klassen einen Konstruktor haben. Da die Erzeugung von Objekten außerhalb der Enum-klasse verboten ist, muss der Konstruktor (oder die Konstruktoren) `private` sein.

Der Aufruf des Konstruktors (die Objekterzeugung) geschieht bei der Deklaration der Enum-Konstanten, indem die Konstruktorparameter durch Argumente der Enum-Konstanten definiert werden.

Beispiel:

```
public enum Color {
    RED(0xff0000), GREEN(0x00ff00), BLUE(0x0000ff),
    WHITE(0xffffff), BLACK(0x000000);

    private int colorValue;

    private Color(int colorValue) {
        this.colorValue = colorValue;
    }

    public int colorValue() {
        return colorValue;
    }
 }
```

Generell werden Enum-Konstanten in der Form *Typ.Wert* also z.B. `Color.RED` angesprochen.[2]

Die switch-Anweisung hat eine etwas besondere Definition. Sie sieht z.B. etwa so aus:

```
switch (color) {
case RED: ...
case GREEN: ...
}
```

Hier entfällt die Typangabe, da der Typ durch die Switch-Variable bereits festgelegt ist.

In geringfügig vereinfachter Form lautet die gesamte Original-Syntax für die Enum-Deklaration:

EnumDeclaration: `enum` *Identifier* `(implements` *TypeList* `)?`
EnumBody

EnumBody: { *EnumConstant* `(,` *EnumConstant* `)*`
`(;` *ClassBodyDeclaration* `)?`
}

EnumConstant: `Identifier ( (` *Arguments* `) )? (` *ClassBody* `)?`

Als wichtiger Punkt fehlt noch, dass hinter einer Enum-Konstanten ein Klassenkörper stehen kann. Während im Normalfall alle Enum-Konstanten Instanzen der gemeinsamen Klasse sind, erreicht man mit dieser Syntax, dass einzelne oder alle Konstanten Instanzen einer jeweils eigenen anonymen Klasse sind. Diese anonymen Klassen sind von der umfassenden Enum-Klasse abgeleitet. So ist es möglich, dass jede Enum-Konstante über ihre eigene Implementierung einer Methode verfügt.

[2] Habe ich irgendwo *static import* besprochen? Dies wird ja dazu verwendet, dass man bei der Verwendung statischer Klassenelemente den Klassennamen weglassen kann. Selbstredend kann *static import* auch die Angabe des Namens der Enum-Klasse erübrigen.

Als Beispiel wollen wir durch die Enum-Klasse `State` die Zustände eines endlichen Automaten beschreiben. Der Einfachheit halber habe dieser Automat nur 2 Zustände $S0$ und $S1$, mit den Übergangen $S0$ nach $S1$ und $S1$ nach $S0$. Zustandsübergänge werden durch die Funktion `nextState()` ausgedrückt. Eine einfache Anwendung kann dann etwa so aussehen:

```
State currentState = State.S0;
...
currentState = currentState.nextState();
...
currentState = currentState.nextState();
```

Vermutlich wird die Klasse `State` ein paar weitere Methoden haben, die hier aber nicht weiter interessieren. Wie sieht nun die Implementierung aus?

```
public enum State {
    S0 {
        public State nextState() {
            return S1;
        }
    },
    S1 {
        public State nextState() {
            return S0;
        }
    };

    public abstract State nextState();
}
```

Bei diesem Idiom ist zu beachten, dass die Methodendeklaration als Bestandteil eines Typs sichtbar ist. Das kann wie hier durch die Deklaration einer abstrakten Methode in der Klasse geschehen, die ja die Oberklasse der anonymen Klassen von $S0$ und von $S1$ ist. Man kann aber auch die Schnittstelle durch ein Interface festzulegen.

```
public interface IState {
    public void nextState();
}

public enum State implements IState {
  S0 { ... }, S1 { ... }  // wie oben
}
```

Merksatz:

Alle Aufzählungsobjekte werden durch die Klasse selbst erzeugt. Sie sind entweder Instanz der Aufzählungsklasse oder einer davon abgeleiteten anonymen Klasse. Die Zuordnung zwischen dem globalen Namen des Objekts und dem Objekt selbst ist unveränderlich. Es ist aber möglich, dass die Objekte ihren Zustand (Inhalt ihrer Instanzvariablen) ändern.

Kapitel 3

Generische Datentypen und Methoden

Typparameter und generische Datentypen wurde bereits in Algorithmen und Programmierung 2 angesprochen. Sie haben gesehen, dass Typparameter einige positive Eigenschaften haben. Dadurch erhält der Compiler erheblich mehr Information über Typbeziehungen. Die Lesbarkeit eines Programms wird so deutlich erhöht. Ein größerer Teil von Typfehlern wird bereits zur Übersetzungszeit erkannt und die meisten Typanpassungen, d.h. Typprüfungen zur Laufzeit werden überflüssig.

Die Kosten für diesen Komfort bestehen in der Anforderung, genauer über die Verwendung von Typbeziehungen nachzudenken und diese in zum Teil komplexen Regeln zu dokumentieren. Die deklarative Seite von Java wird verstärkt.

Typparameter haben im Detail aber auch eine Menge Probleme, die im 2. Semester zunächst ausgeblendet wurden. In diesem Kapitel wird nochmals von vorn angefangen. Typparameter werden zusammenhängend erläutert.

In diesem Kapitel wird mal wieder die Programmiersprache Scala angesprochen. Auch diese enthält Typparameter. Soweit deren Behandlung Java entspricht, wird das nicht weiter thematisiert. Scala enthält jedoch im Detail etwas andere Lösungen. Diese können zu einem größeren Verständnis von Typparametern beitragen.

3.1 Begriffsdefinition

Man nennt die so erweiterten Typen *generische Typen*. Wenn in der Deklaration einer Variablen die bis dahin offenen Parameter festgelegt sind, spricht man auch von einem *parametrisierten Typ*.

Mit *Typ* sind hier Klassen und Interfaces gemeint. Der Begriff `Parameter` bezieht sich darauf, dass Typdefinitionen noch freie Parameter enthalten können, die erst bei der Verwendung ausgefüllt werden. Der Begriff *generisch* drückt aus, dass aus einer allgemeinen Typbeschreibung konkretere Typen erzeugt werden können.[1]

Definition:

*Ein **generischer Typ** enthält in seiner Definition (freie) Parameter. Bei der Verwendung und der Einsetzung eines speziellen Typarguments erhält man daraus*

[1] Generische Typen haben aber nichts mit den Templates von C++ zu tun. In Java geht es nur um den Aspekt der Typprüfung durch den Compiler. In C++ geht es darum, parametrisiert Code zu erzeugen.

*einen **parametrisierten Typ**. Der parametrisierte Typ schränkt die Typdefinitionen von Variablen ein, verhindert damit ungewollte Zuweisung und erlaubt aufgrund des bekannten Typparameters Opertationen, die ohne Typparameter nur mit Cast und Laufzeitprüfung möglich wären. In Java werden Typparameter nur bei der Typprüfung durch den Compiler beachtet. Auf die bei ihrer Erzeugung eigentlich ebenfalls parametrisierten Objekte speichern diese Information aber nicht.*

3.2 Arrays in Java

Java-Arrays haben von Haus aus einen parametrisierten Typ. Sie sollen daher als Erstes besprochen werden. Dabei wird bereits ein wesentlicher Unterschied zu parametrisierten Objekten erkennbar. Trotz der grundsätzlichen Übereinstimmung unterscheiden sie sich in mehreren Punkten von den später eingeführten parametrisierten Typen:

- Für ihre Deklaration gibt es eine besondere Syntax (die von C entlehnt ist).
- Arrayobjekte speichern den Datentyp ihrer Elemente. Bei parametrisierten Typen ist dies (wegen Aufwärtskompatiblität nicht der Fall.
- Obwohl Arrays veränderliche Objekte darstellen, verfügen Sie über die Eigenschaft der Kovarianz (Besprechung folgt weiter unten), die konzeptionell nur bei unveränderlichen Objekten Sinn mach.

Trotz dieser – nur historisch zu begründenden Unterschiede – *sind* Arrays parametrisierte Typen.

Ein Array ist ein Behälter, der eine Menge von Variablen von definiertem Elementtyp enthält. Bei der Erzeugung eines Arrays durch `new`*Typ*`[`*N*`]`, wird durch die Angabe des Elementtyps und der Anzahl der Elemente ein Objekt eines neuen Typs erzeugt. Während die Größe des Arrays ein unveränderliches Merkmal des Objekts ist, spielt sie für den Typ keine Rolle. Entsprechend wird in einer Deklaration neben der Array-Notation nur der Elementtyp angegeben (*Typ*`[]`).

Das folgende Beispiel zeigt zunächst die Vorteile von parametrisiertem Code mit einem Array zu dem (altmodischen) unparametrisierten Code mit einem logisch äquivalenten `ArrayList`.

```
Number[] a = new Number[10];
a[0] = Integer.valueOf(1);
a[1] = 1.5;
a[2] = "hello";   // *** Compilerfehler !!

double s = 0.0;
for (Number x : a)
    s += x.doubleValue();
```

Vergleichen Sie dagegen mal das Beispiel mit einer `ArrayList` aus der Java-Bibliothek (altmodischer Code):

```
// veraltete Variante
List a = new ArrayList(10);
a.add(Integer.valueOf(1));
a.add(1.5);
```

```
a.add("hello");  // *** logischer Fehler wird nicht erkannt

double s = 0.0;
for (Object obj : a) {       // die Liste speichert Object
    Number x = (Number) obj; // Laufzeitfehler ?
    s += x.doubleValue();
}
```

Im Vergleich sollte deutlich werden, wie das ursprüngliche Typkonzept von Java im Vergleich zum Array-Konzept Typprüfungen auf die Laufzeit verlagert. Hier müssen nämlich Casts angegeben werden, die im Prinzip nichts anderes als programmierte Typprüfungen sind. Bei Arrays werden die Typbeziehungen des Beispiels vom Compiler geprüft. Genauso bewirkt dann auch die Verwendung von Typparametern eine vom Compiler garantierte Typsicherheit:

```
List<Number> a = new ArrayList<Number>(10);
a.add(Integer.valueOf(1));
a.add(1.5);
a.add("hello");  // *** Compilerfehler !!

double s = 0.0;
for (Number x : a)      // kein Cast noetig!
    s += a.doubleValue();
```

3.3 Die Verwendung von parametrisierten Typen

Zunächst soll nur die Situation beschrieben werden, die sich bei der parametrisierten Verwendung vorhandener generischer Typen ergibt.[2]

Definition:

Ein parametrisierter Typ *ist ein Referenzdatentyp der zur vollständigen Angabe, neben seinem Namen, weitere Typangaben verlangt. Diese Typinformationen folgen auf den Typnamen und stehen in spitzen Klammern. Der Typname allein, ohne die Angabe der Typparameter, wird als* raw type *bezeichnet.*

Die folgenden Beispiele zeigen die Verwendung.

```
// Objekte
Stack<Character> charStack = new Stack<Character>();
Stack<Integer> intStack = new Stack<Integer>();
ArrayList<String> lst = new ArrayList<String>();

// erlaubte Operationen
charStack.push(Character.valueOf('a'));
intStack.push(Integer.valueOf(3));
lst.add("hello");
for (String s : lst) ...
Integer x = intStack.pop();
int n = lst.get(0).length();

// Compilerfehler
```

[2] Durch die Java-Bibliothek verfügen wir bereits über eine ausreichende Menge von parametrisierten Typen.

```
charStack.push(Integer.valueOf(1));
intStack.push("hello");
lst.add(Character.valueOf('a'));
lst.add(Integer.valueOf(1));
for (Integer a : lst) ...

for (String s : lst) ...
```

Sie sollten versuchen, das Beispiel genau nachzuvollziehen. Beim wirklichen Programmieren würde man hier zum Teil auf Autoboxing zurückgreifen.

3.4 Die Definition von einfachen generischen Klassen

Nachdem wir gesehen haben, wie parametrisierte Klassen verwendet werden, wollen wir uns ansehen, wie man eine generische Klasse definiert. Dabei gilt es zunächst zu erklären, was ein Typparameter ist.

Definition:

*Ein **Typparameter** ist ein Platzhalter für einen Referenzdatentyp. Bei der Deklaration von Variablen und bei der Erzeugung von Objekten muss der Parameter durch einen konkreten Typ oder einen gerade sichtbaren Typparameter belegt werden. Der Typparameter und auch die konkrete Ersetzung stehen jeweils in spitzen Klammern hinter dem Klassennamen. Hat eine Klasse mehr als einen Typparameter, so werden diese durch Komma getrennt aufgelistet.*

Definition:

*Eine **generische Klasse**, ist eine Klasse, die von einem oder mehreren Typparametern abhängt.*

Anmerkung:

Typparameter und generische Klassen sind ausschließlich Konzepte für den Übersetzungsvorgang (Typprüfung). Auch der Classfile enthält noch Informationen über Typparameter, da Classfiles bei der Übersetzung benötigt werden. Zur Laufzeit ist die Parameterinformation und auch die Information über deren aktuelle Belegung nicht vorhanden (type erasure). Eine Konsequenz ist, dass es für jede generische Klasse auch nur ein Klassenobjekt gibt.

Anstelle komplizierter Erläuterungen soll hier einfach das Beispiel einer ganz primitiven Behälterklasse stehen.

```
public class SimpleArray<T> {
    private T[] array;

    // Der Konstruktorname bekommt keine Parameter
    public SimpleArray(T[] array) {
        this.array = array;
    }

    public int length() {
        return array.length;
    }
```

```
    public T get(int index) {
        return array[index];
    }

    public void set(int index, T value) {
        array[index] = value;
    }
}
```

Die Anwendung dieser Klasse geschieht so wie bei den Klassen der Java-Bibliothek.

```
String[] strings = {"hello", "world"};
SimpleArray<String> a = new SimpleArray<String>(strings);
for (String s : a)  ...
```

3.5 Generische Methoden

Methoden können ebenfalls Typparameter einführen. Syntaktisch muss der Ausdruck für die Typparameter vor dem Rückgabetyp der Methode stehen. Bei dem Methodenaufruf werden aber keine aktuellen Typparameter mitgegeben. Diese werden vielmehr durch Typinferenz, d.h. automatisch, ermittelt.

Definition:

*Eine **generische Methode** enthält in ihrer Definition einen oder mehrere Typparameter. Alle freien Parameter müssen in einer durch spitze Klammern begrenzten Liste, die unmittelbar vor der Methodensignatur steht, benannt sein. Bei Aufruf der Methode werden die aktuellen Parameter durch **Typinferenz** durch den Compiler ermittelt.*[3]

Die Klasse `SimpleArray` soll eine Methode besitzen, die mit einem übergebenen Array ein Objekt der Klasse erzeugt und automatisch die richtige Parametrisierung vornimmt.

```
public static <T> SimpleArray<T> create(T[] array) {
    return new SimpleArray<T>(array);
}
```

Der Aufruf kann dann wie folgt aussehen:

```
String[] strings = {"hello", "world" };
SimpleArray<String> a = SimpleArray.create(strings);
```

Als ein weiteres Beispiel nehmen wir an, eine statische Methode kopiere ein Array in ein Objekt, das die Schnittstelle `List<T>` implementiert. Die Deklaration sieht dann so aus:

```
public static <T> List<T> copyToList(T[] array) {
    List<T> result = new ArrayList<T>();
    for (T x: array)
        result.add(x);
```

[3] Die Typinferenz ist der Unifikation in Prolog eng verwandt.

```
    return result;
}
```

Auch bei diesem Beispiel erkennt der Compiler anhand des Übergabetyps `String[]`, dass der Parameter T für `String` steht. Daraus ergibt sich der korrekte Rückgabetyp `List<String>`:

```
String[] stringArray = {"hello", "world"};
List<String> liste = copyToList(stringArray);
```

3.6 Konsequenzen der Typlöschung

Definition:

*Unter dem Begriff **Typlöschung** (engl. **type erasure**) versteht man im Zusammenhang mit parametrisierten Typen den Umstand, dass zur Laufzeit keine Information über aktuell eingesetzte Typparameter zur Verfügung steht.*

Dies hat verschiedene, teils positive, teils eher nachteilige Konsequenzen:

- Generische Klassen definieren nur ein einziges Klassenobjekt. Der Typparameter spielt dabei keine Rolle. Das Klassenobjekt für `ArrayList<String>` und `ArrayList<Integer>` ist `java.util.ArrayList.class`.
- Es können grundsätzlich keine Instanceof-Abfragen gemacht werden, die mit Typparametern zu tun haben.
- Typanpassungen mit Typparametern sind zwar nicht verboten, bringen aber die Warnung mit sich, dass sie nicht geprüft werden. Sie sind trotzdem manchmal notwendig um dem Compiler die nötige Information zu geben.[4]
- Da die Typinformation zur Laufzeit nicht zur Verfügung steht, kann man der Typprüfung bewusst oder unbewusst ausweichen. Das Sicherheitskonzept von Java erfordert daher beim Zugriff auf im Behälter gespeicherte Elemente eine Laufzeitprüfung (diese muss jetzt nur nicht mehr als Cast programmiert werden), obwohl in der Regel der Compiler solche Fehler ausschließen sollte.
- Bei der Erzeugung von Arrays darf ein Typparameter nicht als Elementtyp auftreten, da bei Arrays der Elementtyp immer im Objekt gespeichert ist.

Der letzte Punkt ist manchmal etwas ärgerlich und soll an einem Beispiel verdeutlicht werden. Angenommen, wir wollen eine einfache Stackklasse schreiben. Dann geht das nicht anders als im folgenden Beispiel, das eine Warnung nach sich zieht. In der Praxis wird diese (erwartete) Warnung, so wie auch in diesem Beispiel, durch eine Annotation unterdrückt:

```
public class Stack<T> {
    private T[] array;
    private int top = 0;
```

[4] An dieser Stelle entsteht eine kleine Sicherheitslücke. Aber auch die dabei übersehenen Fehler fallen später irgendwann auf, Trotzdem sollte man sparsam mit solchen Anpassungen umgehen.

```
    @SuppressWarnings("unchecked")
    public Stack(int size) {
        // new T[size] ist nicht erlaubt!
        array = (T[]) new Object[size];
    }

    public void push(T x) {
        array[top++] = x;
    }

    public T pop() {
        T result = array[--top];
        array[top] = null;
        return result;
    }
}
```

3.7 Eingeschränkte Typparameter

Der Typparameter `T` aus dem Stack-Beispiel sagt über die zu speichernden Objekte nichts aus. So gesehen ist er äquivalent zur Typangabe `Object`. Man kann mit Variablen vom Typ `T` nur Methoden aufrufen, die bereits in der Klasse `Object` definiert sind.

Dies ist für reine Behälter wohl in Ordnung, für andere Anwendungen aber oft unzureichend, wie das folgende Beispiel zeigt. Es soll die größte Zahl aus einer Liste ermittelt werden. Mögliche Datentypen sind die Wrappertypen `Short`, `Float`, `Double` und `Long`. Am besten würde man den Algorithmus für den Obertyp `Number` formulieren. Das sähe dann so aus:

```
// wenig sinnvolle Methode
public static Number maxValue(List<Number> lst) {
    Number max = null;
    for (Number x : lst) {
        if (max == null ||
                x.doubleValue() > max.doubleValue())
            max = x;
    }
    return x;
}
```

Diese Definition ist brauchbar, hat aber mehrere Schwächen. Vollständig wird dies bei der Diskussion der Verträglichkeit von parametrisierten Typen weiter unten deutlich. Eine sofort erkennbare Schwäche ist, dass wir nicht erreichen können, dass eine `List<Double` ein Resultat von `Double` zurückgibt. Wir könnten also versucht sein, die folgende Variante zu verwenden, die leider falsch ist.

```
// falsche Realisierung wegen Aufruf von doubleValue()
public static <T> T maxValue(List<T> lst) {
    T max = null;
    for (T x : lst) {
        if (max == null ||
                x.doubleValue() > max.doubleValue())
            max = x;
    }
    return x;
```

```
}
```

Wäre da nicht der Aufruf der Methode von `doubleValue()`, dann wäre das die perfekte Lösung. Sie leistet genau das, was wir wollen. Es fehlt einzig und allein eine zusätzliche Information oder (obere) Beschränkung für den Parameter `T`, dass es sich nämlich unbedingt um einen Untertyp von `Number` handeln muss. Genau dies lässt sich auch angeben:

```
// perfekte Loesung
public static <T extends Number> T maxValue(List<T> lst) {
    T max = null;
    for (T x : lst) {
        if (max == null ||
                x.doubleValue() > max.doubleValue())
            max = x;
    }
    return max;
}
```

Mit der an diesem Beispiel gezeigten Erweiterung erhalten wir die vollständige Syntax für die Deklaration des Typparameters

Die Syntax eines Typparameters sieht wie folgt aus:

$Parameterdeklaration$: $Name$ `(extends` Typ_1 `(&`Typ_i `)*)?`

Die optionale Extends-Klausel beschränkt die für den Parameter zulässigen aktuellen Typen auf Klassen und Schnittstellen die von Typ_1 abgeleitet sind. Typ_1 steht für eine Klasse oder eine Schnittstelle. Die durch `&` getrennte Liste weiterer Typangaben fordert, dass weitere Schnittstellen Typ_i implementiert sind.

3.8 Parametrisierte Typen und Ableitungsbeziehungen

Nach dieser ersten Einführung scheint das Thema Generics (trotz Typlöschung) relativ einfach und damit scheint auch schon alles gesagt zu sein. Dem ist leider nicht so! Die Probleme fangen an, wenn wir uns mit den Fragen beschäftigen, die sich bei der Vererbung ergeben.

3.8.1 Normale Ableitung

Solange sich Ableitungsbeziehungen nicht unmittelbar auf die Typparameter selbst beziehen, ergeben sich keine neuen Regeln.

Definition:

Wenn Typ U ein Untertyp von Typ O ist, dann ist U<X> ein Untertyp von O<X>, wobei X ein beliebiger Typname ist.

Zum Beispiel ist die folgende Zeile korrekt:

```
List<Integer> a = new ArrayList<Integer>();
```

Natürlich gelten die Ableitungsregeln auch bezüglich der eingesetzten konkreten Parameter. Dies wird an dem folgenden Beispielen deutlich:

```
List<Number> a = new ArrayList<Number>();
a.add(Integer.valueOf(5));
Object x = a.get(0);
```

Die Begründung für dieses Verhalten ist auch ganz naheliegend. Der Compiler „weiß“, dass er für alle T's der Schnittstelle `List` einfach ein `Number` einzusetzen hat.

Die immer noch unklare Frage ist, wie es sich mit der Vererbung zwischen den Parametern verhält.

3.8.2 Wie ist es bei Arrays?

Gehen wir jetzt auf die Vererbungsregeln für Arrays ein.

Definition:

Arrays sind wie alle Objekte Untertyp von `Object`*. Als einziger parametrisierter Typ ist ein Array* `U[]` *auch Untertyp eines Array* `O[]`*, wenn* `U` *ein Untertyp von* `O` *ist. Arraytypen verhalten sich* ***kovariant****.*

Diese Definition lässt die folgenden Anweisungen zu:

```
String[] a = new String[10];
Object b = a;
Object[] c = a;
```

Ist hier die Zuweisung `c[0] = Integer.valueOf(8)` erlaubt? Darauf kann es zwei verschiedene Antworten geben. Die eine Antwort kommt vom Compiler, die andere Antwort kommt vom Laufzeitsystem.

- Compiler: Die Zuweisung ist erlaubt, da `c[0]` den Typ `Object` hat.
- Laufzeit: Die Zuweisung ist nicht erlaubt. Eine Zuweisung ist nämlich nur dann erlaubt, wenn das durch `c` referierte Array-Objekt auch einen Elementtyp hat, der ein Obertyp von `Integer` ist (es kommen infrage: `Integer`, `Object`, `Number`, `Comparable`, `Serializable`). Wenn der Elementtyp (hier ist er `String` kein Obertyp von `Integer` ist, liegt ein Typfehler vor.

Man kann sich dieses Verhalten auch so erklären. Ein Array vom Typ `T` stellt eine Menge von Variablen daran (`T a[0]` usw.). Wenn der Compiler den (genauen) Typ nicht kennt, kann er keine Typprüfung vornehmen. Java löst dieses Problem, indem es die Zuweisung zu dem Arrayelement zur Laufzeit prüft. Dies ist leicht möglich, da jedes Arrayobjekt seinen genauen Elementtyp gespeichert hat.

Dieser im Zusammenhang mit der Verwendung von Arrays vom Compiler nicht feststellbare Typfehler, wird zur Laufzeit durch eine `ArrayStoreException` geahndet.

Abgesehen vom Cast-Fehler (`ClassCastException`) ist dies der einzige Typfehler, der nicht vom Compiler erkannt wird.

Den Java-Entwicklern war von Anfang an bewusst, dass sie damit einen Fehler machten, der entgegen der Philosophie von Java nur durch das Laufzeitsystem aus der Welt geschafft werden konnte. Sie nahmen dies wohl in Kauf, um die polymorphe Formulierung von Algorithmen auf Arrays zu ermöglichen.

```
public static void sort(Object[] a);
```

Ohne die Regel der Kovarianz ist diese Methode nutzlus (da man dann nur Arrays von `Object`) sortieren kann. Man müsste für jede Klasse einen eigenen Sortieralgorithmus schreiben. Das Beispiel zeigt, dass parametrisierte Datentypen in einer streng getypten und gleichzeitig objektorientierten Sprache zwingend erforderlich sind.

Nicht parametrisierte Behältertypen bieten natürlich von vornherein keine Typsicherheit. Aber dies führt in der Konsequenz dazu, dass zwischen (altem) Java und einer dynamischen Sprache nur ein gradueller, aber kein grundsätzlicher Unterschied besteht: der Java-Compiler erkennt *einige* Typpfehler, aber nicht alle.

Bei der Einführung von Typparameter mit Java 5 ließen sich die frühen Entwurfsfehler nicht mehr aus der Welt schaffen. Vielmehr bestimmte jetzt die Forderung der Aufwärtskompatibilität das Aussehen der Spracherweiterung. Es wurden dabei zwar eine ganz Reihe von Verbesserungen erreicht, diese wurden aber mit beinahe ebensovielen Schwierigkeiten und Problemen erkauft.

In der Konsequenz haben sich Typparameter in Java nur teilweise durchgesetzt. Nur wenige Java-Programmierer können richtig damit umgegehen.

3.8.3 Ableitungsregeln für Typparameter

Typparameter entsprechen dem Elementtyp von Arrays. Anders als der Elementtyp von Arrays, steht dieser aber nicht zur Laufzeit zur Verfügung. Ist trotzdem eine analoge Vererbungsregel gültig? Die folgende Diskussion zeigt, dass das nicht sein darf.

```
// Problematischer Code
SimpleArray<Integer> aInt = new SimpleArray<Integer>();
SimpleArray<Object> aObj = aInt; // ist das richtig??
```

Wenn die Typregeln für parametrisierte Typen genauso wären wie die für Arrays, wäre dies erlaubt. Wie wir gesehen haben, führte diese Regel bei Arrays aber zur Notwendigkeit einer Laufzeitprüfung (`ArrayStoreException`). Eine solche Prüfung ist aber bei Generics nicht möglich. Da in Java das Prinzip der Typsicherheit (soweit wie möglich zur Übersetzungszeit aber vollständig zur Laufzeit) absolut gilt, kann in dem Beispiel, `aInt` nicht in `aObj` gespeichert werden, ohne die Typprüfung vollständig aufzugeben.

Die bei Array verwendete Regeln ist aber auch konzeptionell fehlerhaft. Man wäre also auch bei Vernachlässigung der Aufwärtskompatibilität nicht die klare Festlegung der Typverträglichkeit herum gekommen.

Definition:

*In Java gilt **Nichtvarianz** von Typparametern: Sei U ein Untertyp von O und T ein beliebiger Typ. T<U> ist* kein *Untertyp von T<O>.*

Schauen wir uns zur Illustration des Problems die Typregeln für die Zuweisung *linke Seite = rechte Seite* an. Es muss gelten, dass der Typ der linken Seite ein Obertyp der rechten Seite ist. Egal welcher Typ links steht, wir können ihn immer durch einen Obertyp ersetzen. Auf der rechten Seite ist es umgekehrt. Egal welchen Typ ein Ausdruck der rechten Seite hat, wir können ihn immer durch einen Ausdruck von einem Untertyp ersetzen. Hinsichtlich der Typverträglichkeit gelten rechts und links vom Zuweisungszeichen entgegengesetzte Regeln.

Als Merkregel können wir uns das so formulieren:

Merksatz:
Für beliebige X ist erlaubt: Obertyp(X) = Untertyp(X)

Da sich für die beiden Seiten einer Zuweisung *entgegengesetzte* Regeln ergeben, dürfen wir die Typverträglichkeit nicht einfach mit der Vererbungsbeziehung gleichsetzen. Wir bleiben dabei, dass die Vererbungsbeziehung der Parameter keine Vererbung zwischen den parametrisierten Typen erzeugt. Weiter unten werden aber Wege gezeigt, wie man genauere Regeln für die Typverträglichkeit angeben kann.

Anmerkung:
Die im Folgenden beschriebenen Regeln sind nicht ganz intuitiv. Dies liegt daran, dass wir im Alltag oft mit etwas anderen Sachverhalten konfrontiert sind. In einem Buch habe ich den folgenden Vergleich gelesen: Wenn das Finanzamt von einem Unternehmen ein Verzeichnis von dessen Beschäftigen erhält, ist das für das Finanzamt gleichzeitig ein Verzeichnis von Personen (Vererbung). Dies gilt aber nur deshalb, weil das Finanzamt nur eine Kopie erhält. Es wäre ein Fehler, wenn das Finanzamt die original Mitarbeiterdatei der Firma bekäme und dort weitere Personen eintragen und so zu Unternehmensmitarbeitern machen könnte. Das Beispiel zeigt bereits: Unveränderliche Objekte sind kovariant!

3.8.4 Verträglichkeitsbeziehungen

Hier soll nun die genauere Angabe der Typverträglichkeit beschrieben werden. Dabei sei U wieder ein Untertyp von O und T ein beliebiger Typ. Wenn wir die Typverträglichkeit parametrisierter Typen in Bezug auf die Vererbung definieren, erhalten wir vier verschiedene Möglichkeiten. Es ist vielleicht erstaunlich, dass jede dieser vier Möglichkeiten in Java ihre wohl definierte und sinnvolle Verwendung findet. Es ist sinnvoll, sich die Namen dieser Varianten zu merken.

Invarianz/Nichtvarianz $T<U>$ und $T<O>$ sind unverträgliche Typen.

Kovarianz $T<U>$ ist zuweisungskompatibel zu $T<O>$. Dieses Verhalten heißt kovariant, da die Regel der Vererbungsbeziehung folgt.

Kontravarianz $T<O>$ ist zuweisungskompatibel zu $T<U>$. Dieses Verhalten heißt kontravariant, da die Regel der Vererbungsbeziehung entgegengesetzt ist.

Bivarianz $T<U>$ und $T<O>$ sind in jeder Richtung miteinander verträglich.

Welche der vier Möglichkeiten im Einzelfall geeignet ist, hängt in Java vom Kontext ab. Der Arraymechanismus realisiert die Kovarianz und löst die damit verbundenen Probleme

durch eine Typprüfung zur Laufzeit. Parametrisierte Typen gehorchen zunächst der Invarianz, d.h. die erzeugten Typen sind nicht verwandt, so dass sich keine Konflikte ergeben können.

Um die nötige Mächtigkeit des generischen Konzepts zu erreichen, hat man Wildcards und Typregeln eingeführt, die gezielt die Formulierung von kovariantem, kontravariantem oder bivariantem Verhalten ermöglichen.

Scala geht den vermutlich konsequenteren (und für den Anwendungsprogrammierer einfacheren Weg). In Scala wird die Varianz direkt bei der Definition eines parametrisierten Typs festgelegt. Der Compiler überprüft, dass keine Operationen möglich sind, die die Typpregeln verletzen.

Wegen der besseren Verständlichkeit soll daher zunächst Scala besprochen werden.

3.9 Typparameter in Scala

In Scala wird die Typverträglichkeit von Typparametern bei der Definition festgelegt. Ein weiterer Vorteil von Scala ist, dass sich auch Arrays hinsichtlich der Syntax und auch in Bezug auf die Typverträglichkeit wie andere parametrisierte Typen verhalten.

Einige „Fehler“ von Java bleiben auch in Java bestehen. Die virtuelle Maschine nebst Kompatilität zu Java erzwingen die Typlöschung und einige Anpassungen an Java.

3.9.1 Grundsätzliche Regeln für Typparameter

Zunächst zur Syntax. In Scala werden Typparameter ähnlich deklariert wie in Java. Sie können bei Klassen, bei Schnittstellen (`trait`) und auch bei Methoden stehen. Der auffälligste Unterschied besteht darin, dass anstelle der spitzen Klammern eckige Klammern verwendet werden.

Weitere Unterschiede bestehen in der größeren Einfachheeit der Scala-Regeln. Insbesondere gilt für Typparameter auch die weitgehende Typinferenz. Arrays werden ebenfalls wie parametrisierte Typen behandelt.

Betrachten wir mal das folgende kommentierte Beispielprogramm:

```
object Anwendung { // object hat keine Typparameter
  def main(args: Array[String]) {   // Array[String] !
    val s = List(1, 2, 3.5)         // s: List[Double]
    val List[Double] = Nil          // Parameter notwendig
    drucke(reverse(s))              // Typinferenz
  }

  def reverse[T](liste: List[T]) = { // Typinferenz
    def rev(l: List[T], r: List[T]): List[T] =
      if (l == Nil) r
      else rev(l.tail, l.head::r)
    rev(liste, Nil)
  }

  def drucke[T](liste: List[T]): Unit =
    for (x <- liste) println(x)
}
```

Wie Sie sehen, sind Typparameter nur dann erforderlich, wenn dem Compiler sonst die

nötige Information fehlt. Es ist auch kein Unterschied in der Behandlung von Array-Typen und anderen Typen erkennbar.

Die folgenden Abschnitte zeigen wie die Varianz bei der Definition des Typs festgelegt wird.

3.9.2 Nicht-Varianz für Typparameter

Typparameter ohne besonderen Zusätze zeigen kein Varianz-Verhalten. Betrachten wir dies an einem Beispiel für eine einfache Datenstruktur, nämlich einen Stack. In dem Beispiel fange ich mit einer normalen Stack-Klasse an. Aufgrund der Anforderungen für Typparameter werden dann aber auch einige Varianten diskutiert.

Zunächst soll eine Schnittstelle für den Stack durch eine abstrakte Klasse beschrieben werden:

```
abstract class Stack[T] {
  def push(x: T): Unit
  def pop: T
  def isEmpty: Boolean
}
```

Als nächstes wollen wir den Stack durch eine (unveränderliche) Liste implementieren:

```
class ListStack[T] extends Stack[T] {
  private var data = List[T]()

  def push(x: T) { data = x::data }
  def pop: T = data match {
    case Nil => throw new NoSuchElementException
    case h::t => data = t; h
  }
  def isEmpty = data = Nil
}
```

Das Stack-Beispiel zeigt die üblichen Regeln der Typinvarianz für Typparameter. Im folgenden ist nur die erste Zuweisung korrekt (die eigentlich unnötigen Typangaben stehen hier um Fehler zu provozieren).

```
val a: Stack[String] = new ListStack[String]  // korrekt
val b: ListStack[Any] = new ListStack[String] // Typfehler
```

3.9.3 Kovarianz für Typparameter

Scala kennt die Notation `[+T]` für die Angabe der Kovarianz. Das Stack-Beispiel lässt sich damit (scheinbar) umformulieren. Zur Vereinfachung verzichte ich jetzt auf die Schnittstelle:

```
class ListStack[+T](private var data: List[T]) {

  /*
  def push(x: T) { data = x::data }  // FEHLER
  */
```

```
  def pop: T = data match {
    case Nil => throw new NoSuchElementException
    case h::t => data = t; h
  }

  def isEmpty = data = Nil
}
```

Wenn wir dies versuchen, wird der Compiler uns bei der Methode `push` einen Fehler melden. Deshalb betrachten wir die Klasse zunächst einmal so, als gäbe es kein `push`.

Ohne `push` können wir keine Elemente zu dem Stack hinzufügen. Das macht von der Anwendung her keinen Sinn. Das Streichen von `push` ermöglicht aber die Korvarianz. `push` würde, wenn es nicht schon vom Compiler verboten wäre, zu Problemen führen.

Damit man die Klasse trotzdem sinnvoll verwenden kann, habe ich einen entsprechenden Konstruktor definiert.

```
val a = new ListStack[Int](List(1,2,3,4))
val b: ListStack[Any] = a    // Kovarianz

val c = b.pop      // c = 1 (Typ von c: Any)
val d = a.pop      // d = 2 (Typ von d: Int)

// wenn jetzt push vom Compiler zugelassen waere:
a.push(3)          // korrekt
a.push("a")        // Compilerfehler und unsinnig
b.push("b")        // zwar "erlaubt" aber unsinnig
```

Die letzte Zeile macht deutlich, warum ein kovarianter Stack kein `push` haben darf. Dies würde nämlich dazu führen, dass dann unsinnige Operationen möglich würden.

3.9.4 Kontravarianz

Kontravariante Typen werden durch ein vorangestelltes Minuszeichen gekennzeichnet. Versuchen wir, auch dies an dem Stack-Beispiel zu erläutern.

```
class ListStack[-T] {

  def push(x: T) { data = x::data }

  /* FEHLER
  def pop: T = data match {
    case Nil => throw new NoSuchElementException
    case h::t => data = t; h
  }
  */

  def print: Unit =
    while (! isEmpty) println(pop)

  def isEmpty = data = Nil
```

Dieses Mal ist es umgekehrt und `pop` ist verboten. `print` soll einen letzten Rest von Brauchbarkeit sichern.

Die Verwendung gestaltet sich jetzt wie folgt:

```
val a = new ListStack[Any]
val b: ListStack[Int] = a // Kontravarianz

a.push("a")  // korrekt
a.push(1)    // korrekt
b.push(2)    // korrekt
b.push("a")  // vom Compiler verboten (wg. Typ von b)

val x: Any = a.pop // immer problemlos
val y: Int = b.pop // kann falsch sein!
```

Das Beispiel zeigt, dass bei Kontravarianz die Verhältnisse umgekehrt zur Kovarianz sind.

3.9.5 Ko- und kontravariante Position

Wie wir gesehen haben, erlaubt die Kovarianz andere Verwendungen als die Kontravarianz. Es geht darum, ob die der Typ sich auf eine Zuweisungsposition (linke Seite einer Zuweisung, Parameterliste) oder auf eine Ausdrucksposition (rechte Seite einer Zuweisung, Rückgabetyp) bezieht. Der Scala-Compiler gewährleistet die strikte Einhaltung dieser Regeln.

Definition:

In der Definition eines kovarianten Typs darf der Typparameter nur in kovarianten Positionen (z.B. Rückgabetyp) auftreten. Umgekehrt darf der Parameter eines kontravarianten Typs nur in kontravarianten Positionen (z.B. Parametertyp) auftreten. Die genauen Regeln sind im Einzelfall nachvollziehbar aber nicht einfach allgemein zu formulieren.

Aus meiner Sicht ist das klarste Beispiel für die unterschiedliche Behandlung von Ko- und Kontravarianz durch die Funktionsschnittstelle gegeben (das steht so in der Scala-Bibliothek):

```
trait Function1[-T,+R] {
  def apply(x: T): R
}
```

So muss es sein. Parameter sind kontravariant, Rückgabe ist kovariant.

Man kann sich das auch anhand von Liskov's Prinzip klar machen. Dieses Prinzip besagt, dass ein Objekt eines Untertyps überall da auftauchen darf wo ein Objekt eines Obertyps erwartet wird. Für Funktionen bedeutet das, dass eine „Unterfunktion" einen größeren Definitionsbereich als die „Oberfunktion" haben darf und dass ihr Wertebereich aber ruhig kleiner sein kann.

3.9.6 Funktionale Datentypen und von unten beschränkter Typ

Das Stack-Beispiel scheint uns vor ein Problem zu stellen? Anscheinend ist hier ausschließlich die Nicht-Varianz möglich. Ähnliche Beispiele haben die Java-Designer wohl zu einer anderen Lösung geführt. Zugegeben, auch in Scala gibt es einen Workaround, den ich aber hier nicht besprechen will.

Da Scala die funktionale Programmierung unerstützt, gibt es eine viel elegantere Lösung. Die grundsätzliche Aussage lautet: *Funktionale Objekte sind immer kovariant*.

Wenn wir das auf den Stack anwenden, müssen wir diesen zunächst in einer unveränderlichen Form schreiben. Veränderungen treten bei `push` und bei `pop` auf. Wir führen eine reine Abfragefunktion `peek` ein, die das oberste Stackelement zurückgibt.[5]

Das neue `pop` gibt einen um ein Element kleineren Stack zurück und `push` gibt einen neuen um ein Element erweiterten Stack zurück. Ein bestehender Stack wird nie verändert.

```
package immutable

class Stack[T] private(private val data: List[T]) {
  // Oeffentlichter Konstruktor
  def this() = this(Nil)

  def peek = data.head
  def isEmpty = data.isEmpty
  def pop = new Stack(data.tail)
  def push(x: T) = new Stack(x::data)
```

Diesen Stack können wir wie folgt anwenden:

```
var s = new Stack[Int]
s = s.push(1)
s = s.push(2)
s = s.push(3)
while (! s.isEmpty) {
  println(s.peek)
  s = s.pop
}
```

Das sieht für jemanden, der prozedurale Programmierung gewohnt ist, etwas umständlicher aus. Aber, wie gesagt, Stack-Objekte sind jetzt unveränderlich.[6]

Es liegt nahe, in der Stack-Klasse jetzt die Kovarianz einzuführen. Doch halt! Formal hat sich nicht viel geändert. In `push` steht `T` immer noch in einer Kovarianz-Position.

Wir können das Problem aber lösen, wenn wir in `push` einen weiteren Typparameter einführen. Was muss hier gelten? Wenn wir mittels `push` ein Element vom Typ `T` oder von einem Untertyp von `T` einfügen, ist der Ergebnisstack immer noch ein `Stack[T]`. Das ist aber nicht unser Problem! Die Kovarianz macht es möglich, dass wir an `push` ein Element eines Obertyps übergeben. Ein solches Element gehört aber nicht in einen `Stack[T]`. Wir müssen den Ergebnistyp als `Stack[U]` beschreiben, wobei `U`, der „niedrigste gemeinsame Obertyp“ von `T` und dem Typ des neuen Elements ist. Die Aufgabe diesen Typ genau herauszufinden, überlassen wir dem Compiler. Wir müssen den Sachverhalt nur genau beschreiben.

Die kovariante Stack-Klasse sieht jetzt so aus:

```
package immutable

class Stack[+T] private(private val data: List[T]) {
```

[5] Eine andere Lösung würde als Ergebnis von `pop` ein Paar von Wert und neuem Stack zurückgeben.

[6] Da dies ein prozedurales Programm ist, taucht hier `var` auf. In einer funktionalen Stackanwendung würde dies verschwinden.

```
  // Oeffentlichter Konstruktor
  def this() = this(Nil)

  def peek: T = data.head
  def isEmpty: Boolean = data.isEmpty
  def pop: Stack[T] = new Stack(data.tail)

  def push[U >: T](x: U): Stack[U] = new Stack[U](x::data)
}
```

Die entscheidende Beziehung lautet: `U >: T` der Typ `U` kann ein beliebiger Obertyp von `T` sein.[7] Im Beispiel sieht das so aus

```
var s = new Stack[Int]   // s: Stack[Int]
s = s.push(1)
s = s.push(2)
// s = s.push("3")       // verboten !!
var b = s.push(2.0)      // b: Stack[AnyVal]
var c = b.push("a")      // c: Stack[Any]
```

Die Typregeln stellen sicher, dass ein gegebener Stack nur Elemente enthält, die mit dem Stacktyp verträglich sind.

3.10 Beschreibung von Verträglichkeitsbeziehungen in Java

Nachdem wir Scala ausführlich betrachtet haben, kehren wir zu Java zurück. Wie schon gesagt, ist das Konzept der Typparameter dasselbe, nur dass in Java die Varianz nicht bei der Definition, sondern bei der Verwendung des Parameters festgelegt wird.

3.10.1 Die exakte Typangabe

Zunächst gilt auch in Java das Prinzip der Nicht-Varianz

Beispiel:

```
List<Number> numberList = new ArrayList<Number>();

// Es ist erlaubt abgeleitete Typen von Number zu verwenden.
numberList.add(Double.valueOf(3.5));

// Listen als Ganzes sind nur mit List<Number> vertraeglich.
List<Number> numberList2 = numberList;
List<Double> doubleList = numberList; // FEHLER
List<Object> objectList = numberList; // FEHLER
```

Die Schnittstelle `List` definiert auch eine Methode `addAll`, die bewirkt, dass die Inhalte einer anderen Datensammlung der Liste hinzugefügt werden. Leicht modifiziert könnte die Schnittstelle so aussehen.

```
public interface List<T> extends Collection<T> {
    ...
    public void addAll(Collection<T> other);
```

[7] `U` darf auch gleich `T` sein. Wenn ich von Unter- und Obertyp spreche, ist dies immer mitgemeint.

```
}
```

Anmerkung:

Die Tatsache, dass der Typ `Collection` *auftaucht, stellt kein Problem dar, da er ein Obertyp von* `List` *ist.*

Vergleichen Sie das folgende Beispiel. Die erste Methode, nämlich Elemente eines Untertyps mittels `add` einzeln einer Liste hinzuzufügen, ist erlaubt. Die zweite Methode, die das gleiche bewirkt, ist es jedoch nicht. Das ist eine Folge der zu eingeschränkten Spezifikation von `addAll`. Weiter unten wird die richtige und bessere Deklaration gezeigt.

```
List<Number> numberList = ...
List<Double> doubleList = ...

// erlaubtes Hinzufuegen
for (Double x : doubleList)
    numberList.add(x);

// Typfehler !
numberList.addAll(doubleList);

// ebenfalls Typfehler
List<Number> var1 = doubleList;
List<Number> var2 = (List<Number>) doubleList;
```

3.10.2 Unbeschränkter Wildcard

Als Platzhalter für einen beliebigen Typ kann das Fragezeichen `?` verwendet werden. Damit ist nicht mehr über die Objekte ausgesagt, als dass sie vom Typ `Object` sein können. An so deklarierten Behältern sind keine Veränderungen möglich.

Merksatz:

Der unbeschränkte Wildcard sagt aus, dass es völlig egal ist, wie der Typ parametrisiert ist. Entsprechend wenig darf man mit den Objekten tun.

Der Typparameter darf bei der Verwendung einer entsprechend deklarierten Variablen keine Rolle spielen. Diese Lösung genügt dem Prinzip der Bivarianz.

Beispiel:

```
public static void printList(List<?> list) {
    for (Object x : list)
        System.out.println(x);
}
```

Die Klassenfunktion `printList` darf mit Objekten eines beliebig parametrisierten Listentyps aufgerufen werden. Die folgenden Aufrufe sind korrekt:

```
ArrayList<Integer> alst = ...
LinkedList<String> llst = ...
printList(alst);
printList(llst);
```

Der Preis, den man für diese Flexibilität zahlt, besteht darin, dass innerhalb von `printList` keine Veränderungen an dem Listenobjekt erlaubt sind. Erlaubt ist nur das Lesen von Inhalten. Dabei ist keine Typinformation über die Inhalte vorhanden, so dass der Typ `Object` angegeben ist.

Diese Anwendungen des unbeschränkten Wildcard sieht man immer dann, wenn es auf den konkreten Typ der Inhalte nicht ankommt. Schauen wir uns auch noch ein Beispiel an.

```
List<Number> numberList = ...

// Das ist verboten:
List<Object> objList1 = numberList; // FEHLER

// Das ist erlaubt:
List<?> objList2 = numberList;
List<?> weitereVariable = objList2;
Object x = objList2.get(0);

// Das ist verboten, egal was wir uebergeben:
objList2.add( ...);
```

3.10.3 Von oben beschränkter Wildcard

Wird die Typangabe gemäß `? extends` Typ_1 `&` Typ_i beschränkt, spricht man von einem von oben beschränkten Wildcard. Diese Variante des Wildcard ist mit dem (von oben) beschränkten Typparameter verwandt. Hinsichtlich der Verträglichkeit ergibt sich die Regel der Kovarianz. Die Typregeln stellen sicher, dass keine verbotenen Operationen erfolgen können. So kann keine Zuweisung zu den Elementen eines so parametrisierten Behälters erfolgen.

Merksatz:

Der von oben beschränkte Wildcard sagt aus, das es sich bei dem aktuellen Typparameter um den angegebenen oder einen davon abgeleiteten Typ handeln muss. Diese Information steht dem Compiler zur Verfügung. Es sind nur solche Informationen erlaubt, die mit dieser Information verträglich sind.

Beispiel

```
public static double doubleSumme(
        List<? extends Number> lst)
{
    double s = 0;
    for (Number x: lst)
        s += x.doubleValue();
     return s;;
}
```

Hier können wir erneut die bereits angesprochene Methode `addAll` aufgreifen. Die Deklaration dieser Methdoe in Java-Bibliothek lautet nämlich:

```
public interface List<T> extends Collection<T> {
    ...
```

```
    public void addAll(Collection<? extends T> other);
}
```

Schauen wir uns ein paar Beispiele an:

```
List<Number> numberList = ...
List<Double> doubleList = ...

// das ist jetzt erlaubt:
numberList.addAll(doubleList);

// das ist mit Recht verboten:
doubleList.addAll(numberList); // FEHLER

// Auch Zuweisungen sind erlaubt:
List<? extends Number> var = doubleList;

// die Elemente von var gelten als Number
Number n = var.get(0);
Double x = (Double) var.get();

// var erlaubt kein add
var.add(Double.valueOf(4,3)); // FEHLER
```

3.10.4 Von unten beschränkter Wildcard

Wird die Typangabe gemäß `? super` Typ_1 `&` Typ_i beschränkt, spricht man von einem von unten beschränkten Wildcard. Hinsichtlich der Verträglichkeit ergibt sich Kontravarianz.

Merksatz:

Der von unten beschränkte Wildcard sagt aus, das es sich bei dem aktuellen Typparameter um den angegebenen oder einen Obertyp davon handeln muss. Diese Information steht dem Compiler zur Verfügung. Es sind nur solche Operationen erlaubt, die mit dieser Information verträglich sind.

Typische Beispiele sind die Deklaration von `Comparable` und `Comparator`.

Eine naheliegende Deklaration wäre:

```
public <T> T maxObject(List<T> lst, Comparator<T> c) {
     T max = null;
     for (T x: lst) {
         if (max == null || c.compareTo(x, max) > 0)
             max = x;
     }
     return max;
}
```

Diese Lösung hat aber wieder die übliche Einschränkung der Typverträglichkeit. Betrachten Sie das folgenden Anwendungsbeispiel, das so nicht funktioniert. Es geht dabei darum für alle Unterklasse von `Number` (etwa `Integer` und `Double`) einen gemeinsamen Comparator zu schreiben.

```
public class NumberComparator implements Comparator<Number> {
    public int compare(Number a, Number b) {
        double x = a.doubleValue();
        double y = b.doubleValue();
        return (x < y) ? -1 : (x > y) ? +1 : 0;
    }
}

List<Double> lst = ...

Double n = maxObject(lst, new NumberComparator());
```

Von der Ausführung her, wäre das Beispiel lauffähig. Allerdings lässt es sich nicht übersetzen, da der Compiler zunächst in der letzten Zeile beim Aufruf von `maxObject` folgert, dass für den Typparameter T gemäß der Deklaration von `lst` der Typ `Double` zu verwenden ist. Andererseits erfordert der Typ des Comparator-Objekts den Typ `Number`. In diesem Fall gibt es keine Lösung, da wegen der Invarianz exakter Typangaben kein mit beiden Parametern verträglicher Typ existiert.

Wir benötigen also eine allgemeinere Typangabe, die uns erlaubt, auch `Comparator`-Objekte zu verwenden, die für Obertypen von T definiert sind.

```
public <T>
T maxObject(List<T> lst, Comparator<? super T> c) {
    T max = null;
    for (T x: lst) {
        if (max == null || c.compare(x, max) > 0)
            max = x;
    }
    return max;
}
```

Diese Lösung erfüllt genau ihren Zweck. Exakt das gleiche gilt auch für die Angabe, dass die Listenelemente von sich aus `Comparable` sein sollen. Hier haben wir einen komplexeren Typausdruck, der dann auch deutlich macht, wo die Grenzen der Typbeschreibung liegen:

```
public <T extends Comparable<? super T> >
T maxObject(List<T> lst) {
    T max = null;
    for (T x: lst) {
        if (max == null || x.compareTo(max) > 0)
            max = x;
    }
    return max;
}
```

Ein letztes Beispiel, das aber hoffentlich nicht verwirrt, soll verdeutlichen, dass die Beschränkung von unten im Unterschied zur Beschränkung von oben, größere Möglichkeiten bei Zuweisungen eröffnet:

```
List<Number> numberList = ...
List<? extends Number> obenBeschraenkt = numberList;
List<? super Number> untenBeschraenkt = numberList;
```

```
// lesender Zugriff
// genauester Typ!
Number num = obenBeschraenkt.get(0);
Object obj = untenBeschraenkt.get(0);

// schreibender Zugriff
obenBeschraenkt.add(...); // FEHLER
untenBeschraenkt.add(Double.valueOf(3.4));
```

Beim lesenden Zugriff auf den von unten beschränkten Typ-Parameter der Liste, wissen wir praktisch nichts über den Typ des Ergebnisobjekts. Natürlich kann es ein `Double` oder ein `Integer` sein, aber letztlich wissen wir nur mit Sicherheit, dass der Typ von `Object` abgeleitet ist.

Der schreibende Zugriff ist dagegen unproblematisch. Wir wissen dass mindestens alle Untertypen von `Number` an die Liste angehängt werden können. Das gilt wenn es sich um eine `Number` handelt. Die Liste könne aber auch als eine `List<Object>` definiert sein. Darin dürfen wir sogar beliebige Objekte speichern. Mit Sicherheit ist also das Anhängen eines `Double`-Objekts oder eines anderen Objekts, das einen Untertyp von `Number` hat, zulässig.

3.11 Sonstige Bemerkungen zu Typparametern in Java

Im Folgenden sind noch zwei eher technische Anmerkungen zu Typparametern aufgeführt. Sie sind nur ganz kurz dargestellt. Für Details verweise ich auf die Java-Sprachbeschreibung.

3.11.1 Zunehmende Typinferenz in Java

Wie Sie wissen, wird in Scala überall wo möglich, bei Bedarf der genaue Datentyp vom Compiler ermittelt, wenn er nicht schon im Programm angegeben ist. Diesen Mechanismus nennnt man *Typinferenz*.

Java 5 und Java 6 kennen Typinferenz bereits beim Aufruf von Methoden. Ein Beispiel ist das folgende Szenario:

```
public static <T> T maxValue(Collection<T> liste, Comparator
   <? super T> c);

...

List<Person> persons = ...
Person longTallSally = maxValue(persons, heigthComparator);
```

Der Compiler erkennt, dass er für `T` den Typ `Person` einsetzten muss.

In Java 7 kam ein weiterer Anwendungsfall hinzu. Bei der Erzeugung von Objekten kann der Parameter entfallen, wenn er sich aus dem Typ der Zuweisung ergibt.

```
List<Person> persons = new ArrayList<>();
```

Um deutlich zumachen, dass man nicht den alten Still (ohne Typparameter) verwendet, muss der Konstruktor aber mit den <> „geschmückt“ sein.

Durch Java 8 wird Java bekanntlich um Lambda-Ausdrücke erweitert. Die in Java gewählte Form der Parametrisieren wäre hier extrem umständlich. In der Konsequenz gilt bei Lambda-Ausdrücken ebenfalls Typinferenz. Gleichzeitig wird die Verwendung von Lambda-Ausdrücken ausdrücklich auf die Fälle eingeschränkt, in dem der Typ für den Compiler erkennbar ist.

```
double total = myList.stream().reduce(0, (x,y) -> x+y);
```

3.11.2 Capture Conversion

Das ist schon eine etwas spezielle Regel. Sie bezieht sich auf einen Sonderfall des Aufrufs einer generischen Methode, in der der aktuelle Typ durch einen Wildcard ausgedrückt ist. Ein typisches Beispiel ist dann gegeben, wenn ein Typparameter nur dazu nötig ist, die Beziehung zwischen einem Übergabetyp und einem Rückgabetyp oder auch zu dem Typ einer lokalen Hilfsvariable auszudrücken. Ein Beispiel ist der folgende Code für das Umdrehen der Reihenfolge der Daten einer Liste:

```
public static void reverse(List<?> liste) {
    rev(liste);
}

private static <T> void rev(List<T> liste) {
    List<T> temp = new ArrayList<T>(liste);
    int last = liste.size() - 1;
    for (int i = 0; i <= last; i++)
        liste.set(i, temp.get(last - i));
}
```

Die Motivation für den Wildcard in der Signatur von `reverse` ist dadurch gerechtfertigt, dass es grundsätzlich möglich sein sollte, jede Liste, auch solche, deren Elementtyp man nicht kennt, umzudrehen. Die Capture Conversion besteht darin, dass die Methode `rev` aufgerufen werden kann, obwohl ihr Aufruf eigentlich einen bestimmten Parametertyp verlangt.

3.11.3 Aufwärtskompatibilität zu altem Code

Die Regeln für generische Typen sind so definiert, dass die Aufwärtskompatibilität zu altem Code garantiert ist. Das heißt, es macht kein Problem, wenn alter Code neue generische Klassen nutzt. Wenn im umgekehrten Fall neuer Code alte Klassen nutzt, wird dies auch funktionieren. Dies ist jedoch grundsätzlich unsicher, da dabei die striktere Typprüfung des Compilers ausgehebelt wird.

```
class Alt {
    public static void addData(List liste) {
        liste.add(Integer.valueOf(1));
        liste.add(Integer.valueOf(2));
    }
}

class Neu {
    void method() {
        List<String> strings = new ArrayList<String>();
```

```
        Alt.addData(strings); // Warnung: type safety
        String erster = strings.get(0); // ClassCastException
    }
}
```

Wegen der Aufwärtskompatibilität muss erlaubt sein, dass die Methode `Alt.addData` aufgerufen wird. Der Compiler kann dann aber nicht mehr garantieren, dass nur Strings in die Liste eingefügt werden. Der Aufruf ist erlaubt, aber ungeprüft. Der Compiler gibt an dieser Stelle eine Warnung aus. Die Unsicherheit bedeutet, dass der Compiler bewusst oder unbewusst hintergangen werden kann. Damit solche Typfehler am Ende doch noch erkannt werden, setzt der Compiler von sich aus Typprüfungen ein, die bei vollkommen sicheren Code nicht nötig wären.

Die Aufwärtskompatibilität hat eine weitere Konsequenz, die auch vollständig neuen Code nicht ganz ohne unsichere Operationen und die damit verbundenen Warnungen auskommen lässt. Es ist nämlich, wie bereits oben angesprochen, nicht möglich, Arrays mit einem Typparameter zu erzeugen, da aus Kompatibilitätsgründen die Information über Typparameter zur Laufzeit nicht vorliegt.

3.11.4 Die Lösung des Array-Problems in Scala

Der letzte Abschnitt zeigt, dass es innerhalb des Java-Typsystems erhebliche Brüche gibt. Das ist, neben der hohen formalen Komplexität des Systems der Typparameter mit ein Grund warum heute schon viele Experten der Meinung sind, dass Java die Grenzen seiner Weiterentwicklung erreicht hat. Dies ist ein Schicksal, das wohl jedes lebendige Softwaresystem mit der Zeit ereilt.

Es ist instruktiv, sich zum Vergleich die relativ neue Scala-Lösung anzusehen. Zunächst einmal, auch Scala kann hier nicht perfekt sein. Es baut ja schließlich auf der JVM auf und muss so auch mit deren Limitierungen leben. Auch Scala ist von der Typlöschung betroffen!

Die Scala Lösung erinnert an das in Java verbreitete Muster der *reification* (zu deutsch: Verdinglichung). Damit ist gemeint, dass Typinformation explizit (in Form von Klassenobjekten) gespeichert wird. Als Beispiel diene hier eine „normale“ Stack-Klasse auf Array-Basis:

```
class Stack[T: ClassManifest] {
  private val data = new Array[T](100)
  private var top = 0

  def isEmpty = (top == 0)
  def push(x: T) {
    require(top < 100)
    data(top) = x
    top += 1
  }

  def pop() = {
    require(top > 0)
    top -= 1
    val result = data[top]
    data[top] = null.asInstanceOf[T] // garbage collector!
    result
  }
}
```

Die „Typangabe“ beim Typparameter ist eine Kurzschreibweise dafür, dass dem Konstruktor implizit ein weiteres Argument übergeben wird und zwar ein vom Compiler automatisch generiertes Manifest-Objekt für den Parametertyp. Man erkennt, dass so durch den Scala-Compiler die Typlöschung umgangen wird.

Kapitel 4

Nebenläufigkeit

4.1 Grundbegriffe

Ein sequentielles Computerprogramm bestimmt den Ablauf der Programmschritte und damit die genaue Reihenfolge interner und externer Aktionen. Der festgelegte Ablauf erleichtert das Verständnis eines Algorithmus und die Fehlersuche ganz erheblich.

Ein vollständig festgelegter Programmablauf hat aber auch Nachteile:

- Flexible Benutzerschnittstellen sind nur möglich, wenn Programmabläufe nicht vollständig spezifiziert sind.
- Moderne Hardware enthält mehrere Ausführungskerne. Ihre effiziente Ausnutzung ist ohne Nebenläufigkeit nicht möglich.
- Effiziente Serveranwendungen erfordern die Unterbrechung zeitaufwändiger Operationen zugunsten kurzfristiger Aktionen.

Parallele und quasiparallele Arbeitsweise kennt man nicht nur zwischen verschiedenen Prozessen sondern auch innerhalb eines Prozesses. Dieser Fall, in dem es um eine besonders flexible Art der Programmierung geht, die von Java explizit unterstützt und für einige Anwendungen auch unbedingt benötigt wird, soll hier näher besprochen werden.

Definition:
*Ein **Programm** beschreibt statisch die zur Lösung eines Problems durchzuführenden Aktionen. Ein **Prozess** ist ein in der Ausführung befindliches Programm.*

Definition:
*In einem **sequentiellen Prozess** finden alle beobachtbaren Aktionen in der zuvor durch das Programm festgelegten Reihenfolge statt. Der Compiler, die virtuelle Maschine und die Rechnerhardware sind frei, diese Reihenfolge zu variieren, soweit das keine Auswirkungen auf den beobachtbaren Ablauf hat.*

Definition:
*Die Ausführung eines **nebenläufigen Programms** besteht aus mehreren sequentiellen Abläufen. In welcher Reihenfolge diese Abläufe bearbeitet werden, oder ob sie sogar gleichzeitig ausgeführt werden, ist durch das Programm nicht festgelegt.*

Findet die Ausführung gleichzeitig statt, so spricht man auch von paralleler Programmausführung. Sonst spricht man von einer quasiparallelen Ausführung des Programms. Der allgemeinere Begriff ***Nebenläufigkeit*** *umfasst beides.*

Bei nebenläufigen Programmen ist die Reihenfolge der Aktionen nicht vollständig festgelegt. Ein korrektes nebenläufiges Programm muss so geschrieben sein, dass in allen möglichen Ausführungsvarianten immer ein korrektes Verhalten zustande kommt.

Die Ausführung eines Prozesses wird durch das Betriebssystem unterstützt. Dabei werden dem Prozess Betriebsmittel, wie Speicher, Rechenzeit und Ein-/Ausgabemedien zugeteilt.

Definition:

Ein ***Thread*** *ist ein einzelner sequentieller Ausführungsstrang.*[1] *Bei nebenläufiger Ausführung kann er innerhalb eines Prozesses gleichzeitig oder quasigleichzeitig mit anderen Threads ausgeführt werden (Multithreading). Ein Thread greift auf den Adressraum und auf die Betriebsmittel seines Prozesses zu. Gleichzeitig verfügt jeder Thread mit dem Aufrufstack der Methoden und den in dessen Stackframes gespeicherten lokalen Variablen über lokalen Speicher. Die Rechenzeitzuteilung zu ausführungsbereiten Threads nennt man* ***Scheduling****. Sie erfolgt in der Regel durch das Betriebssystem.*

4.2 Grundbegriffe der Parallelverarbeitung

Die gleichzeitige Ausführung eines Programms auf mehreren Rechnern oder Rechnerkernen gewinnt immer mehr an Bedeutung. Es ist daher angebracht, den weiteren Überlegungen zu Multihreading in Java, das ja in der Regel nur einen Computer betrifft, einige Begriffe aus dem Bereich der Parallelverarbeitung voranzustellen.

4.2.1 Parallele Rechnerarchitekturen

Zunächst sind ein paar Sätze zur Rechnerhardware angebracht. Hier gibt es verschiedene Maßstäbe zur Beurteilung von Parallelität.

Man kann zunächst Parallelität mit unterschiedlicher *Granularität* finden. So gibt es in jedem Prozessor Parallelität auf *Wortebene* (frühe Computer arbeiteten tatsächlich teilweise bitseriell). Außerdem verfügen moderne Prozessoren über komplexe Rechenwerke, die mehrere Befehle überlappend ausführen können (*Pipelining*).

Neben einer solchen Parallelität auf Instruktionsebene kann es dann Parallelität auf großeren Einheiten, wie Threads oder Prozessen geben.

Bereits 1966 veröffentlichte Michael Flynn ein Schema, genannt *Flynn'sche Taxinomie* mit dem man ganz grob unterschiedliche Parallelrechner charakterisieren kann (vgl. Abb. 4.1). Der Taxinomie liegt zugrunde, dass ein (von Neumann-) Rechner gesteuert durch einen Kontrollfluss (*incstruction*) Daten (*data*) verändert. Zu jedem Zeitpunkt kann es einen oder mehrere Kontrollflüsse und ein oder mehrere gleichzeitig bearbeitete Datelemente geben. Mit den Abkürzung M für *multiple*, S für *single* sowie I und D für *instruction* und *data*. Ergeben sich vier denkbare Kombinationen. SISD ist also ein Kontrollfluss, der jeweils ein Datenelement bearbeitet beschreibt den herkömmlichen sequentiell arbeitenden Computer. Während MISD keine Bedeutung spielt stehen MIMD und SIMD für unterschiedliche Klassen der Parallelverarbeitung.

[1] = Faden, Wortherkunft germ. drat

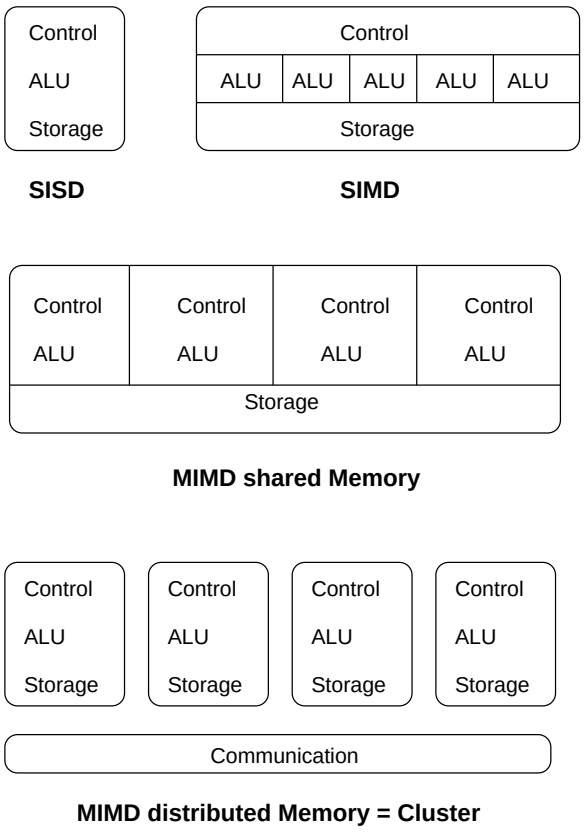

Abbildung 4.1: Flynn'sche Klassifikation und Speicherorganisation von Parallelrechnern.

SIMD-Rechner (*single instruction multiple data*) zeichnen sich dadurch aus, dass sie über einen einzigen Befehlsstrom verfügen. Dieser operiert gleichzeitig auf einer großen Menge von Daten. Typische Vertreter dieser Kategorie sind die Vektorrechner. Bei einer Addition zweier Vektoren können dann alle Vektorkomponenten gleichzeitig addiert werden. Durch die Verwendung eines einzigen Steuerwerks ergibts sich eine hohe potentielle Einsparung. Gleichzeitig gestaltet sich sowohl Programmierung als auch Rechnerarchitektur besonders einfach.

Die andere große Klasse der Parallelrechner sind die MIMD-Rechner (*multiple instructions multiple data*). Hier hat man mehrere Prozessorkerne, die mehrere Befehlsströme gleichzeitig (auf unterschiedlichen Daten) ausführen. MIND-Rechner unterscheiden sich grundsätzlich in den Varianten Rechner mit gemeinsamem globalem Speicher und Rechner mit lokalem Speicher. Die weit verbreiteten Rechnercluster gehören in die letzte Kategorie.

Einen Sonderfall stellen Rechner mit mehreren Prozessorkernen dar. Diese Entwicklung der letzten Jahre erfolgte, weil es nicht mehr sinnvoll möglich war den Prozessortakt weiter zu steigern. Abgesehen von speziellen Anwendungen (Webserver) stellt die Programmierung dieser Rechner eine bisher nicht bewältigte Herausforderung dar. In dieser Vorlesung werden einige Ansätze angesprochen.

Den unterschiedlichen Architekturmodellen entsprechen auch ähnliche Programmierkonzepte. Dem SIMD-Prinzip entspricht die Ausnutzung *paralleler Datenstrukturen*, dem MIMD-Prinzip mit gemeinsamem Speicher entspricht die Threadprogrammierung mit ge-

meinsamen Variablen (Java Primitive) und dem MIMD-Prinzip mit verteiltem Speicher entspricht der Botschaftenaustausch (z.B. Scala Actors).

Es ist zu beachten, dass hinsichtlich der Speicherorganisation keine Übereinstimmung zwischen Hardware und Software bestehen muss. Auf gemeinsamem Speicher können Botschaftskonzepte implementiert werden und ebenso kann auf verteiltem Speicher ein virtueller gemeinsamer Speicher realisiert werden. Darüber hinaus jeder Rechner jede der Flynn'schen Klassen emulieren.

4.2.2 Interaktion von parallelen Prozessen

Programme mit mehreren Ausführungssträngen (Nebenläufigkeit, Multiprocessing, Multithreading) enthalten in ihrem Ablauf ein hohes Maß von Unbestimmtheit (*Nichtdeterminismus*). Eine effiziente Ausnutzung paralleler Hardware kann nämlich nur erreicht werden, wenn die unterschiedlichen Abläufe nicht ständig aufeinander abgestimmt werden müssen und wenn sie sich nicht gegenseitig im Ablauf hemmen.

Dies führt aber zu dem Problem, dass auch die Ergebnisse der Programmausführung unbestimmt werden können. Diese (in der Regel unerwünschte) Abhängigkeit der Ergebnisse von den Zufälligkeiten der Programmausführung nennt man *Wettlaufbedingung* oder *race condition*.

Zur geregelten Interaktion von verschiedenen Abläufen gibt es grundsätzlich zwei unterschiedliche Möglichkeiten. Zunächst einmal ist es möglich, kontrolliert über *Botschaften* zu kommunizieren. Das bietet sich von selbst an, wenn die kommunizierenden Prozesse auf unterschiedlichen Rechnern ablaufen, die ihrerseits über Nachrichtenkanäle kommunizieren.

Auf eng gekoppelten Systemen mit gemeinsamem Speicher (z.B. Multicore-Rechner) ist es jedoch meist effizienter direkt über gemeinsame Variable Information auszutauschen. Es gibt verschiedene Verfahren auch dabei Wettlaufbedingungen zu vermeiden. Diese Verfahren bilden die Grundlage des Multithreading in Java.

Die aktuelle Erfahrung besagt, dass das Multithreading über gemeinsamen Speicher so komplex und fehleranfällig ist, dass es in der Regel als nicht bewältigt angesehen werden muss. Eine besondere Eigenart ist, dass Fehler der Nebenläufigkeit kaum durch Tests erkannt werden können. Dies bürdet der Fachkenntnis und der Programmierdisziplin des Entwicklers ein zu hohe Aufgabe auf. In der Konsequenz wird Multithreading gerne vermieden (mit allen damit verbundenen Nachteilen). Besser ist es, man geht konsequent in Richtung der sichereren Verfahren des Botschaftenaustauschs. Dieser Weg wird auch von Scala unterstützt.

4.2.3 Geschwindigkeitszuwachs durch mehrere Prozessoren

Naiv würde man erwarten, dass bei gleichzeitiger Ausführung eines Programms durch N-Prozessoren die Ausführungszeit um einen Faktor N verringert wird. Davon ist man in der Realität jedoch meist weit entfernt. Zunächst benötigt die Verwaltung der Nebenläufigkeit, der Datenaustausch und die Synchronisation zwischen den parallelen Abläufen einen oft nicht unbeträchtlichen Laufzeitaufwand. Dazu kommt, dass in sehr vielen Fällen die Algorithmen selbst nur unvollständig parallelisierbar sind.

Definition:

T_1 bezeichne die Rechenzeit die die Lösung eines Problems auf einem Prozessor

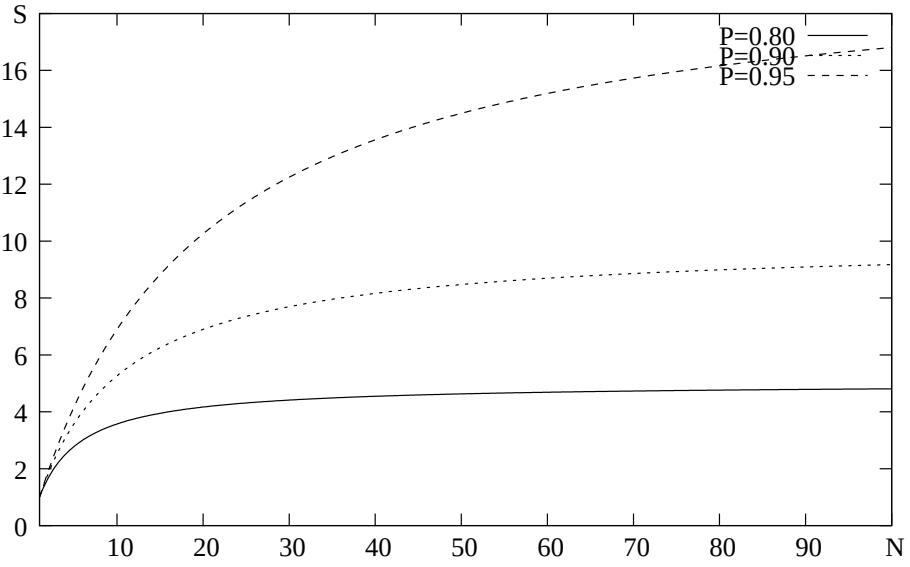

Abbildung 4.2: Amdahls Gesetz: Speedup S in Abhängigkeit der Prozessorzahl N.

*erfordert, T_N die Rechenzeit, die bei N-Prozessoren nötig ist. Der **Geschwindigkeitsgewinn (speed up)** S wird als Verhältnis dieser Zeiten bestimmt: $S = T_1/T_N$. Die Effizienz der Parallelität (**Prozessorausnutzung**, **efficiency**) $\eta = T_1/(NT_N)$.*

Das (naive) Ziel ist, dass $T_N = T_1/N$ ist. Damit ist $S = N$ und für die erwartete Effizienz gilt $\eta = 1$. Dies nennt man auch *linearen speed up*.

In aller Regel wird man diese optimistischen Werte jedoch nicht erreichen können. Am bekanntesten ist in diesem Zusammenhang die pessimistische Abschätzung durch *Amdahl's law*. Gene Amdahl, einer der führenden Mainframe-Produzenten, wollte damit wohl nachweisen, dass sich Parallelverarbeitung kaum lohnt.

Amdahls Gesetz (s. Abb. 4.2) geht davon aus, dass ein Computerprogramm aus einem parallelisierbaren Anteil und einem sequentiellen Anteil besteht. Sei P der relative Anteil des parallelisierbaren Teils. Dann gilt für den Laufzeitgewinn:

$$S = T_1/T_N = \frac{1}{(1-P) + P/N}$$

Für den Fall $N \to \infty$ ergibt sich als asymptotischer Wert für die Laufzeitverringerung der Wert $1/(1-P)$. Das ist z.B. für den Wert $P = 0.9$ eine bescheidene Zunahme um den Faktor 10. Die skeptische Ansicht kann davon ausgehen, dass nur wenige Anwendungen eine Parallelisierung von 90 % des Codes ermöglichen. In der Konsequenz scheint sich Parallelverarbeitung kaum zu lohnen. In dem Beispiel mit $P = 0.9$ erreicht man mit 10 Prozessoren eine Leistungssteigerung von $5,3$. Die erreichte Rechnerausnutzung ist dann gerade 53 %. Will man durch Parallelität dem Grenzwert 10 noch näher kommen, sinkt die Ausnutzung schnell weiter. Die doppelte Prozessoranzahl erreicht gerade den Faktor $6,9$ und eine Effizienz von nur noch 34 %.

Abbildung 4.3 zeigt eine nicht ganz so pessimistische Darstellung des Gesetzes. Hier wird nämlich bei gegebener Prozessoranzahl die Parallelisierbarkeit als Parameter angesehen.

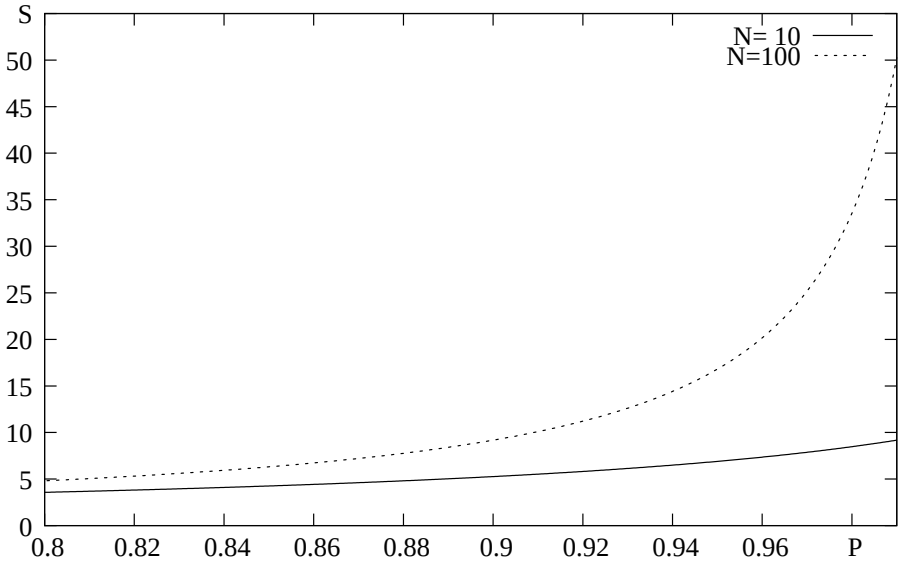

Abbildung 4.3: Amdahls Gesetz: Speedup S in Abhängigkeit der Parallelisierbarkeit P.

Die Darstellung zeigt. dass ein Rechner hoher Parallelität (nur) sinnvoll von hochgradig paralleler Software ausgenutzt werden kann.

Sie können in Wikipedia auch gerne den Begriff „Gustavsons Gesetz“ nachschauen. Es beschreibt den optimistischen Fall, dass mit zunehmender Problemgröße der parallele Anteil mit der Prozessoranzahl wächst, so dass ein linearer Gewinn erzielt wird.

Für unsere Zwecke lautet das Fazit, dass bei „normalen“ Anwendungen durch Nebenläufigkeit nur ein geringer Laufzeitvorteil erzielbar ist. Die tatsächlichen Vorteile sind in diesem Bereich eher softwaretechnischer Natur. Leider werden diese Vorteile aufgrund ungenügender Sachkenntnis und unzureichender Programmiermethodik bisher nur selten erreicht.

Für Probleme höchster Rechenleistung (Simulation der Galaxienentwicklung, Klimamodelle, Strömungsberechnungen usw.) werden Rechnersysteme mit Zehntausenden bis hin zu einer Million Prozessoren eingesetzt.

4.2.4 Abgrenzung zu Multithreading im engeren Sinne

Wie der letzte Abschnitt andeutete, eröffnet sich bei dem Versuch der Leistungssteigerung durch Parallelverarbeitung ein ganz neues Gebiet möglicher Probleme und möglicher Lösungen. Dieses soll hier nicht weiter betrachtet werden.

Hier geht es primär um Anwendungen, die auf einem System mit einem oder wenigen Prozessoren innerhalb eines einzigen Betriebsystemprozesses laufen. Dies bedeutet einerseits eine starke Einschränkung gegenüber *Verteilten Systemen*, andererseits bedeutet es für die Programmierung, dass es eine Vielzahl von möglichen Lösungen für die Implementierung der Nebenläufigkeit gibt.

In der Vorlesung versuche ich klar zu machen, dass der in Java auf Sprachebene implementierte Ansatz der Nebenläufigkeit in der Originalform für praktische Anwendungen ungeeignet ist. Allerdings ist es auch so, dass moderne GUI- und Webanwendungen

nicht ohne Nebenläufigkeit denkbar sind. Wir werden die Mechanismen der Kooperationen zwischen gleichzeitigen Abläufen besprechen, ihre Probleme verstehen lernen und einige Ansätze für einen erheblich besseren Umgang mit Nebenläufigkeit kennenlernen.

4.3 Implizite Parallelität

Wir werden im folgenden sehen, dass das Programmieren von Nebenläufigkeit nicht ganz einfach ist.

Damit ergibt sich ein Dilemma. Die Hardwareentwickler sind momentan nicht mehr in der Lage schnellere Prozessoren zu entwickeln und kompensieren dies damit, dass sie auf einem Chip mehrere Prozessorkerne unterbringen, die die geringere Taktfrequenz durch parallele Operationen ausgleichen sollen.

Die Softwareentwickler sind gewohnt immer schnellere Prozessoren zu bekommen, so dass sie sich keine Gedanken über die Ausnutzung der Leistungsfähigkeit machen müssen. Sie sind kaum in der Lage, parallel arbeitende Programme zu schreiben.

Es gibt natürlich kein Generalkonzept. Die Entwickler von Programmiersprachen haben aber erkannt, dass für sehr viele Anwendungsfälle die funktionale Programmierung eine extrem einfache Lösung anbietet. Letzten Endes ist das der Grund, dass funktionale Programmierung, die über Jahrzehnte (als zu ineffizient) nicht beachtet wurde, heute (aus Effizienzgründen) modern ist.

Im folgenden sollen zwei Ansätze kurz darstellt werden. Zunächst das Fork-Join Konzept von Java 7. Dieses kommt noch ohne funktionale Programmierung aus, zeigt aber den grundlegenden Mechanismus. Anschließend die Lösung, die heute schon in Scala und in Zukunft auch in Java 8 verfügbar ist.

4.3.1 Datenparallelität

Der beispielhafte Anwendungsfall für implizite Operationen sind Vektoroperationen auf Datenstrukturen[2]

Ein primitives Beispiel ist die Addition zweier gleichlanger Arrays. Grundsätzlich hat dieser Algorithmus die Komplexität $O(n)$.

```
for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
}
```

Der Körper der For-Schleife kann grundsätzlich in einem Schritt ausgeführt werden, wenn wir über n Prozessoren verfügen. Oder aber, wenn wir nur p Prozessoren haben, wenigstens in nur n/p Schritten.

Voraussetzung für die Parallelität ist die Unabhängigkeit der einzelnen Schritte. Ein Gegenbeispiel ist ein Filteralgorithmu, der jedes Element durch die Summe der bisherigen Elemente ersetzt (Partialsumme):

```
public static void partialSum(double[] a) {
    for (int i = 1; i < a.length; i++) {
```

[2] In den 80er und 90er Jahren waren alle Supercomputer Vektorrechner, die ihre Leistungsfähigkeit aus dem SIMD-Prinzip bezogen.

```
        a[i] = a[i] + a[i-1];
    }
}
```

Hier muss nämlich bei jeder Zuweisung darauf gewartet werden, dass die vorhergehende Zuweisung (für ein um 1 kleiners `i`) beendet ist. Der Ablauf ist zwingend sequentiell. Das ändert sich allerdings auch nicht, wenn wir die schönere funktionale Lösung wählen:

```
public static double[] partialSum(double[] a) {
    double[] p = new double[a.length];
    p[0] = a[0];
    for (int i = 1; i < a.length; i++) {
        p[i] = a[i] + p[i-1];
    }
}
```

Es gibt aber immerhin Algorithmen, in denen eine ganz einfache Parallelisierung möglich ist. In anderen Fällen muss mehr Aufwand getrieben werden, oder man muss, wenn möglich, auf ganz neue Algorithmen ausweichen.

Für die einfachen Fälle gibt es die ganz einfache Lösung funktionaler Datenstrukturen. Für viele andere Fälle ist die allgemeinere aber auch schon deutlich aufwändigere Fork-Join Lösung brauchbar.

4.3.2 Fork-Join Framework

Die Entwicklung der Nebenläufigkeit ist in hohem Maße von einer einzigen Person, nämlich von Prof. Doug Lea, geprägt. Das von ihm vorgeschlagen Frameword orientiert sich and der Divide-And-Conquer Idee der Algorithmenentwicklung.

Wenn wir einen Algorithmus in zwei (oder mehr) unabhängige Teile aufsplitten können, so können die auch nebenläufig ausgeführt werden. Sind diese immer noch zu groß können wir sie rekursive weiteruntereilen. Wir können dabei wählen, ob die durch Rekursion entstehenden Teilalgorithmen innerhalb eines Threads sequentiell abgearbeitet werden oder ob sie durch mehrere Threads/Prozessoren parallel ausgeführt werden. Wichtig ist, dass die Ergebnisse am Ende wieder zusammengeführt werden. Daraus ergibt sich der Name: *fork* = „aufteilen“ und *join* = „zusammenführen“.

Das klassische Beispiel für diesen Prozess sind Sortieralgorithmen. Meiner Meinung nach hat das dazu geführt, das Fork-Join *vor allem* für das Sortieren geeignet ist. In Wirklichkeit spielt Sortieren für die Parallilisierung eine fast vernachlässigbare Rolle. Ich will daher einen (zwar auch nicht typischen) einfacheren Algorithmus wählen: die Addition aller Zahlen eines Arrays.

```
class Adder extends RecursiveTast<Double> {
    private final double[] a;
    private final int i0;
    private final int n;
    private static final int LIMIT = 1000;

    public Adder(double[] a, int i0, int n) {
        this.a = a;
        this.i0 = i0;
        this.n = n;
    }
```

```
    @Override // implementiert abstrakte Methode
    protected Double compute() {
        if (n > LIMIT) {
           Adder a1 = new Adder(a, i0, n/2);
           a1.fork();
           Adder a2 = new Adder(a, i0+n/2, n-n/2);
           return a2.compute() + a1.join();
        } else {
            double s = 0;
            for (int i = i0; i < n; i++)
                s += a[i];
            return s;
        }
    }
```

Diese Beispiel macht den Fork-Join Charakter deutlich: Zunächst wird die Berechnung mit `a1.fork()` in einen separaten Thread verlagert und am Ende wird für die Zusammenfassung der Teilergebnisse mit `a1.join()` auf das Ergebnis gewartet.

Das Konzept ist eng mit der rekursiven Grundidee verwandt. Einfache Probleme werden direkt – hier: sequentiell – gelöst. Komplexe Probleme werden aufgeteilt – hier: parallel ausgeführt.

Es bleibt anzumerken, dass das Framework in Wirklichkeit nicht bei jeder Aufteilung einen neuen Thread startet. Das wäre extrem ineffizient. Vielmehr wird auf einen Pool bereits vorhandener (oder evtl. doch erst zu startender) Threads zurückgegriffen. Dadurch wird das Starten einer nebenläufigen Aktivität extrem beschleunigt.

Das Framework ist zudem offen für programmierte Optimierungen.

4.3.3 Parallele Datenstrukturen

Der letzte Abschnitt zeigt, dass die rekursive Aufteilung einer Aufgabe oft möglich und auch nicht allzu kompliziert ist. Trotzdem geht es oft noch erheblich einfacher.

Beispiele dafür sind Scala (es geht auch mit `sum`):

```
val sum = a.par.fold(0) ((x,y) => x+y)
```

oder in Java 8:

```
double sum = a.parallel().reduce(0, (x,y) -> x+y)
```

Die Grundidee ist, dass wir unserer Datenstruktur eine parallele Version zuordnen und die parallele Durchführung einem bereits optimierten Bibliotheksalgorithmus überlassen. Es ist davon auszugehen, dass die Ausführung intern mittels Fork-Join durchgeführt wird.

Abgesehen von der kürzeren Schreibweise sind so auch sicher weniger Programmierfehler und Performanefehler möglich. Aber natürlich beschränkt diese Konzept die Parallesierung auch auf einen der (relativ vielen) speziellen Anwendungsfälle.

Nach meiner Meinung ist diese Art der Parallelisierung mittelfristig sehr hilfreich. Langfristig werden (wie man bereits aus den 80ern weiß), sich aber auch die Grenzen der reinen Datenparallelität immer mehr zeigen.

Jedenfalls ist die Möglichkeit der Datenparallelität für die Entwickler der Java-Bibliothek der Hauptgrund für die Einführung funktionaler Elemente in Java 8 und dann später in Java 9.

4.4 Threadzustände

Ein Thread kann während seiner Lebensdauer die Zustände *erzeugt*, *bereit*, *ausführend*, *wartend* und *beendet* annehmen (die folgenden Erläuterungen beziehen sich auf Java), Der Zusammenhang dieser Zustände ist in Abbildung 4.4 dargestellt. Dort ist der exakte Vorgang des Wartens illustriert, der momentan noch nicht so detailliert besprochen wird:

- Nach der Erzeugung des Threadobjekts ist der Thread zunächst in dem Zustand *erzeugt*. Das Objekt ist zwar vorhanden, es können auch Methoden des Threadobjekts von anderen bereits existierenden Threads aufgerufen und ausgeführt werden, der neue Thread wurde jedoch noch nicht gestartet.
- Nach dem Starten eines Threads mittels der Methode `start()` wird der Thread in den Zustand *bereit* versetzt. Nach dem Start befindet sich der Befehlszeiger des Thread vor der ersten Anweisung der Methode `run()`.
- Ein ausführungsbereiter Thread kann von dem Scheduler für die Ausführung ausgewählt und damit in den Zustand *ausführend* versetzt werden. Das Scheduling kann in Java durch `setPriority()` beeinflusst werden. Die genaue Schedulingstrategie aber von der jeweiligen Laufzeitumgebung (Betriebssystem, virtuelle Maschine) ab.
- Die Ausführung eines Thread kann durch den Scheduler jederzeit unterbrochen werden. Damit wird der Thread wieder in den Zustand *bereit* zurückversetzt. Die genaue Strategie der Unterbrechung hängt von der Laufzeitumgebung ab. Man spricht von einem *preemptiven Scheduling* (vorzeitige Unterbrechung).
- Wenn ein Thread auf Bedingungen warten muss (`sleep()` und `wait()`) wird die Ausführung unterbrochen. Der Thread wird in den Zustand *wartend* versetzt. Er ist nicht mehr ausführungsbereit. Eine Zuteilung von Rechenzeit kann erst wieder erfolgen, nachdem er durch ein externes Signal (z.B. die interne Uhr) in den Zustand *bereit* versetzt wurde. Wenn ausschließlich diese erzwungene Art der Unterbrechung realisiert ist, spricht man auch von einem *kooperativen Scheduling*, da dann der Threadwechsel nur durch explizite Anweisungen innerhalb des Threads selbst zustande kommt.
- Erreicht ein Thread das Ende der Methode `run()`, wird er in den Zustand *beendet* versetzt. Das Threadobjekt besteht weiter, seine Methoden (auch diejenigen die sich auf den Threadzustand beziehen) können von anderen Threads weiter aufgerufen werden. Ein erneuter Start des Thread ist aber nicht möglich!

Anmerkung:

Häufig findet man in Programmen von Anfängern, dass die Methode `yield()` *in der Absicht aufgerufen wird, die aktuelle Threadausführung zu unterbrechen und andere Threads zur Ausführung zu bringen. Dies ist ein Programmierfehler! Bei* `yield()` *handelt es sich um einen bloßen Hinweis an den Scheduler, dass andere Threads momentan bevorzugt werden könnten. Die virtuelle Maschine braucht diesen Hinweis nicht zu beachten.*

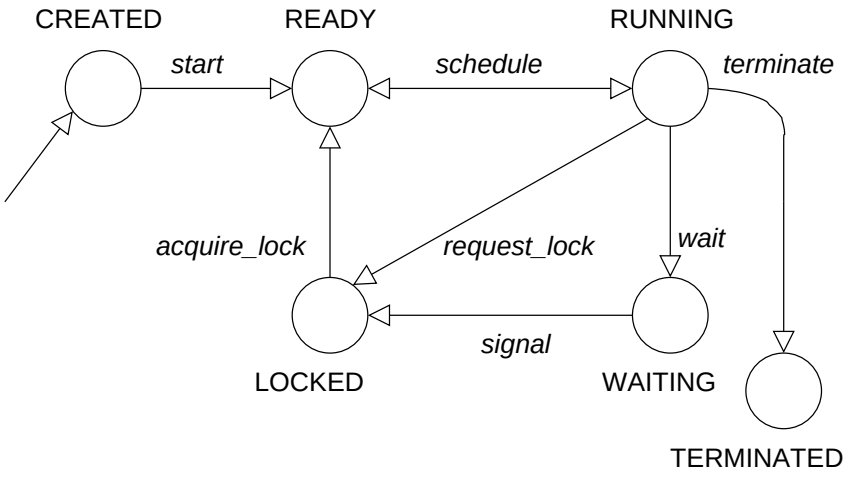

Abbildung 4.4: Threadzustände

Eine Anwendung wird erst dann beendet, wenn alle Threads, die nicht als Dämon[3] deklariert wurden, beendet sind. Ein Dämon-Thread wird dadurch gekennzeichnet, dass vor seinem Start seine Methode `setDemon(true)` aufgerufen wird. Das Ende von `main()` beendet den main-Thread, aber daher nicht unbedingt die Programmausführung. Wenn man das sofortige Programmende erzwingen will, kann man `System.exit()` aufrufen.

Nicht aufgefangene Unterbrechungen beenden den laufenden Thread, nicht das Programm.

4.5 Gemeinsame Variable und Wettlaufbedingungen

Prozesse können nur mithilfe des Betriebssystems kommunizieren. Threads kommunizieren oft nur über gemeinsame Variable.[4] Im Unterschied zu höheren Formen der Kommunikation erhofft man sich durch den direkten Zugriff auf gemeinsame Variable einen deutlichen Effizienzvorteil. Wir sollten uns die von Java angebotenen Möglichkeiten also genauer ansehen.

In Java gibt es hinsichtlich des Threadingverhaltens verschiedene Arten von Variablen:

Lokale Variable. Lokale Variable werden auf dem Stack abgelegt. Jeder Thread besitzt einen eigenen Stack, so dass auf lokale Variable nur die aktive Methode zugreifen kann.

Parameter. Funktions- und Methodenparameter verhalten sich wie lokale Variable.

Threadlokale Variable. Diese werden durch die Klasse `ThreadLocal` erzeugt und verwaltet. Sie wurden eingeführt um Variablen zu haben, auf die man global überall zugreifen kann, deren Sichtbarkeit und Veränderbarkeit sich aber jeweils nur auf

[3] Ein Dämon (englisch *demon*) tut was man ihm sagt und stört sonst nicht weiter.

[4] Es ist guter Stil, wenn gemeinsame Variable in besonders definierten Kommunikationsobjekten versteckt sind.

einen einzigen Thread bezieht. Wenn in unterschiedlichen Threads derselbe globale Name verwendet wird, sind damit unterschiedliche Speicherbereiche und Inhalte gemeint.

Instanzvariable. Die Variablen eines Objekts sind auf dem Heap gespeichert. Bei dem Heap handelt es sich um einen Speicherbereich, der allen Threads gemeinsam ist. Daher können mehrere Threads auf die Inhalte eines Objekts zugreifen. Dies schafft die Voraussetzung für die Kommunikation zwischen Threads. Der Zugriff auf gemeinsame Variable ist jedoch gleichzeitig das mit Abstand größte Problem der Java-Programmierung! Ein Grund dafür ist, dass einzelne Threads lokale Kopien der Werte von Instanzvariablen haben können, deren Inhalte nicht immer miteinander übereinstimmen. Ein anderes Problem besteht in der Interferenz bei der Veränderung von Objektzuständen.

Klassenvariable. Klassenvariable verhalten sich genauso wie Instanzvariable. Es ist manchmal nützlich sie sich als Instanzvariable des Klassenobjekts vorzustellen.

4.5.1 Wettlaufbedingungen

Die Objektorientierung fordert, dass der Zugriff zu Instanzvariablen und zu Klassenvariablen nur über die Methodenschnittstelle erfolgt. Unter anderem soll dadurch gewährleistet werden, dass außerhalb der Methoden der Klasse immer ein gültiger Objektzustand vorliegt. Dies ist der Inhalt des Konzepts der *Klasseninvariante*.

Für nebenläufige Programme sind diese Prinzipien noch wichtiger als sonst. Gleichzeitig ist ihre Einhaltung jedoch stärker gefährdet. Nehmen wir an, ein Thread ist gerade dabei, die Methode eines Objekts auszuführen. Dabei wird er zeitweise die Invariante verletzen, da ja er nicht mehrere Variablen gleichzeitig ändern kann. In einem sequentiellen Programm ist das in Ordnung, da andere Methoden erst aufgerufen werden können, wenn die laufende Methode fertig ist und das Objekt wieder gültig ist. Bei Multithreading kann es aber vorkommen, dass ein zweiter Thread während der Methodenausführung dieselbe oder eine andere Methode desselben Objekts aufruft. Wenn die Methode das Objekt dabei in einem ungültigen Zwischenzustand vorfindet, wird sie nicht mehr korrekt funktionieren. Der Programmierer kann nicht wissen, was passiert, da ihm ja der genaue zeitliche Ablauf und die Abfolge der Befehle unbekannt sind. Da solche Ereignisse von der zufälligen Rechenzeitzuteilung abhängen, nennt man sie *Wettlaufbedingung* oder *race condition*.

Definition:

*Eine **Wettlaufbedingung** (**race condition**) liegt vor, wenn mehrere Threads unkoordiniert auf gemeinsame Variable zugreifen, wobei wenigstens ein Thread den Inhalt verändert, und dadurch das Ergebnis des Programms vom Scheduling abhängt.*

Merksatz:

Der Programmierer muss dafür sorgen, dass der Zugriff auf gemeinsame Variable koordiniert wird.

Das folgende Beispiel verdeutlicht das Problem von Wettlaufbedingungen.

```
@NotThreadSafe
public class Counter {
    private int count = 0;
```

```
    public void count() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

Ein Teil der Probleme hängt mit der unten besprochenen Sichtbarkeit von *Variableninhalten* zu tun.[5] Der offensichtlichere Anteil hängt aber an der Operation `count++`. Dazu muss man wissen, dass diese Operation innerhalb der virtuellen Maschine oder auf der Hardwareebene in aller Regel nicht atomar, (nicht unterbrechbar), abgearbeitet wird.

Man kann sich Ausführung von `count++` gemäß dem folgenden Programmstück vorstellen:

```
1  int register = count;
2  register++;
3  count = register;
```

Als Beispiel habe `count` den Inhalt 3. Zwei Threads A und B rufen beide die Methode `myCounter.count()` auf. Es ist wichtig, dass es sich hier beide Male um dasselbe Objekt handelt. Als erster gelangt Thread A bis kurz vor die Ausführung von Zeile 3. Der Inhalt seiner lokalen Variablen `register` ist jetzt 4. Der Scheduler unterbricht jetzt den Thread und führt den Thread B aus. B findet in der Variablen `count` immer noch den Inhalt 3 vor. Er erhöht die Zahl um 1 und speichert das Ergebnis 4 in `count` ab. Jetzt kommt irgendwann Thread A wieder an die Reihe. Er fährt da fort, wo er unterbrochen wurde und speichert seinen lokalen Wert 4 in `count`. Insgesamt haben wir zweimal die Variable `count` um 1 erhöht. Im Ergebnis hat sich damit der Wert von `count` jedoch nur um 1 verändert. Das ist offensichtlich falsch.

Sie können einwenden, dass ich einen ungünstigen Ablauf ausgewählt habe und dass bei einem anderen Ablauf das richtige Endergebnis herausgekommen wäre. Da haben Sie recht. Aber genau das ist das Problem von Wettlaufbedingungen, dass wir nicht wissen, was herauskommt. In den meisten Fällen wird das Ergebnis vielleicht sogar stimmen. Das ist besonders schlimm! Dadurch können wir nebenläufige Programme nur ganz schlecht testen!

Die Klasse `Counter` ist trotzdem schön, gut und korrekt solange sie nur in einem einzigen Thread genutzt wird. Sobald mehrere Threads darauf zugreifen, ist ihre Funktion jedoch nicht mehr vorhersagbar. Daher habe ich die Klasse mit der Annotation `NotThreadSafe` gekennzeichnet-

4.5.2 Sichtbarkeit

Während die eben erläuterte Ursache für Wettlaufbedingungen für Programmierer leicht nachvollziehbar ist, gibt es weitere Ursachen, die leider allzu häufig ignoriert werden, nämlich die automatische Optimierung der Programmausführung durch Compiler, virtuelle Maschine und Prozessor.

Es wäre blauäugig, sich auf den Standpunkt zu stellen, Optimierungen zu verbieten, die eventuell den nebenläufigen Ablauf eines Programms gefährden. Damit würde man einen

[5] Das ist nicht zu verwechseln mit der durch `public` usw. ausgedrückten Sichtbarkeit der *Variablen*.

großen, vielleicht den größten, Teil der Performancesteigerungen von Computerhardware wieder zunichte machen. Schnelle Prozessorhardware lässt sich nur wirksam ausnutzen, wenn die Nachteile der sequentiellen Arbeitsweise des Prozessors und des langsamen Zugangs zum Hauptspeicher beseitigt oder wenigstens abgemildert werden.

In dem von Neumann'schen Computermodell holt sich der Prozessor seine Werte aus dem Hauptspeicher und schreibt nach einer Operation die Ergebnisse dorthin zurück. Da die Zugriffszeiten zum Hauptspeicher erheblicher länger sind als die Zykluszeiten der Prozessoren ist man dazu übergegangen, Werte in Caches zwischenzuspeichern oder direkt in CPU-Registern zu belassen. Je nach Caching-Strategie finden direkte Zugriffe zum Hauptspeicher nur noch relativ selten statt.

Für sequentielle Programme ist das Verhalten von Caches vollkommen transparent. Dies kann auch in der Nebenläufigkeit so sein, zumindest dann, wenn der Rechner nur über einen einzigen Prozessor oder Prozessorkern verfügt. Es kann aber auch sein, dass ein Prozessor nicht mitbekommt, wenn ein anderer Thread (über einen anderen Prozessor) Daten verändert, so dass er nicht mehr auf dem letzten Stand ist. Für nebenläufige Programme ergibt sich die Aufgabe, dafür zu sorgen, dass der Zugriff auf gemeinsame Variable immer den aktuell gültigen Wert liefert.

Merksatz:

Nebenläufige Programme müssen dafür sorgen, dass gemeinsame Variable und Objekte im aktiven Thread den aktuell gültigen Inhalt haben.

Als Beispiel möchte ich nochmals auf die Klasse `Counter` vom letzten Abschnitt zurückkommen. Nehmen wir an, dass der Thread A mehrmals den Zähler erhöht, so dass der Inhalt von `count` gleich 7 ist. Es kann jetzt sein, dass anschließend der Thread B als Ergebnis von `getCount()` die Rückgabe 2 (oder irgendeinen anderen Wert von 0 bis 7) erhält. B hat vielleicht etwas früher schon mal `Count()` aufgerufen, sodass das Objekt in seinem damaligen Zustand in seinem lokalen Cache gespeichert wurde.

4.5.3 Umordnung von Befehlen

Ein weiteres Problem hat mit der Optimierung der CPU zu tun. Alle modernen Prozessoren verfügen über mehrere Rechenwerke und über die Möglichkeit, mehrere Operationen gleichzeitig, u.U. im Pipelining-Betrieb auszuführen. Um die Ausführungsgeschwindigkeit zu optimieren, können Compiler, virtuelle Maschine, oder einfach in letzter Instanz der Prozessor Änderungen in der Reihenfolge von Operationen vornehmen solange dies in einem sequentiellen Prozess keine beobachtbaren Auswirkungen hat.

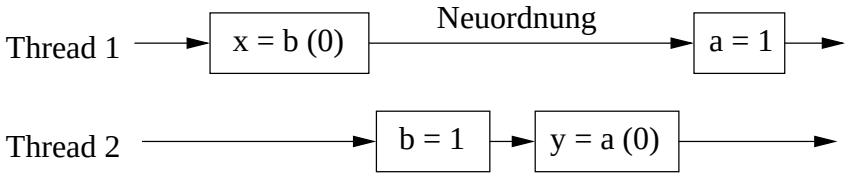

Abbildung 4.5: Umordnung von Anweisungen

Das Problem besteht nun darin, dass bei nebenläufiger Ausführungen diese Änderungen der Ausführungsreihenfolge beobachtbar sein können. Dies führt natürlich dann zu nicht

gewollten Resultaten. Auch hier wird die Verantwortung für die korrekte Ausführung nebenläufiger Programme auf den Programmierer verlagert, der dafür sorgen muss, dass die notwendigen Reihenfolgebedingungen eingehalten werden.

Merksatz:

Nebenläufige Programme müssen dafür sorgen, dass Variablenwerte einer Ausführungsreihenfolge entsprechen,, die mit der Programmlogik vereinbar sind. (happens-before Relation).

```
@NotThreadSafe
public class PossibleReordering {
    static int x = 0, y = 0, a = 0, b = 0;

    public static void main(String[] a) throws Exception {
        // Die Erzeugung zweier Threads.
        // (Wird weiter unten besprochen.)
        Thread t1 = new Thread() {
            public void run() {
                a = 1;              // 1.1
                x = b;              // 1,2
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                b = 1;              // 2.1
                y = a;              // 2.2
            }
        };

        // Start der Threads
        t1.start(); t2.start();

        // Warten auf das Ende der Threads
        t1,join(); t2.join();

        System.out.println("(" + x + "," + y + ")");
    }
}
```

Die Frage ist, welche Ergebnisse wir hier für `x` und `y` erwarten.Da nur die Werte 0 und 1 vorkommen, gibt es theoretisch maximal vier verschiedene Möglichkeiten. Für die Wertepaare (x, y) sind die die Belegungen (0,0), (0,1) (1,0) und (1,1) denkbar. Aber welche davon können bei diesem Programm auch herauskommen? Die erstaunliche Antwort ist: alle vier!

Wir können verschiedene Abläufe durchgehen. Zur Veranschaulichung habe ich die vier entscheidenden Anweisungen nummeriert. 1.2 bedeutet dabei, dass der Thread 1 die Anweisung 2 seiner Methode `run()` ausführt.

- Ein Ablauf ist die Folge 1.1, 1.2, 2.1, 2.2. In dem Fall muss (0,1) herauskommen.
- Der nächste Fall ist umgekehrt 2.1, 2.2, 1.1, 1,2. Es kommt (1, 0) heraus.
- Die Threads können überlappen: 1.1, 2.1, 2.2, 1.2. Wir erhalten (1,1).

Und wie kann es zu (0,0) kommen? Dieses Ergebnis ist möglich, wenn die Reihenfolge der Befehlsausführung geändert wurde. Innerhalb eines sequentiellen Threads ist nicht zu

beobachten, ob die Reihenfolge 1.1, 1.2 ist oder umgekehrt 1.2, 1.1. Entsprechend verhält es sich auch mit den Anweisungen von Thread 2. Wir können aber nicht wissen, was der Compiler und die CPU letztlich gemacht haben.

Wir erhalten als vierten Fall 1.2, 2.2, 2.1, 1.1 mit dem Ergebnis (0,0).[6] In Abbildung 4.5 ist eine Grafik des Geschehens zu sehen.

Sie dürfen natürlich nicht erwarten, dass Sie durch einem Test wirklich alle Ergebnisse reproduzieren können. Meine Aussage ist nur, dass alle grundsätzlich möglich sind.

4.6 Starten und Beenden von Threads in Java

In Java müssen alle Threads die Schnittstelle `Runnable` und damit auch die Methode `run()` implementieren. Threads werden erzeugt und gesteuert über die Klasse `Thread`. Da die Klasse Thread selbst `Runnable` implementiert, ergeben sich zwei Möglichkeiten der Threaderzeugung.

4.6.1 Threaderzeugung mittels Vererbung

Das eine Verfahren besteht darin, eine Klasse von der Klasse `Thread` abzuleiten und `run()` zu überschreiben:

```
public class MyThread extends Thread {
    private int instanzVariable;

    public MyThread(String threadName, int parameter) {
        super(threadName);
        instanzVariable = parameter;
    }

    public void run() {
        // Hier stehen meine Anweisungen fuer den Thread-
           Ablauf
        weitereMethode();
    }

    public void weitereMethode() {
        // Tue was.
    }

    public static void main(String[] args) {
        Thread t1 = new MyThread("Thread-A", 7);
        Thread t2 = new MyThread("Thread-B", 8);
        t1.start();
        t2.start();
    }
}
```

In `main()` werden zunächst zwei unterschiedliche Objekte einer Klasse erzeugt. Wie bei anderen Klassen auch, kann ich über den Konstruktor Argumente an das neue Objekt übergeben. Der Konstruktor initialisiert das Objekt; er startet noch nicht einen besonderen

[6] Man könnte auf die Idee kommen, dass (0,0) auch dadurch herauskommt, dass der main-Thread nicht die letzten Werte der Variablen sieht. Das ist aber nicht richtig. `join()` bewirkt, dass die durch den Thread vorgenommenen Änderungen publiziert werden. Ohne `join()` wäre allerdings (0,0) auch wegen fehlender Sichtbarkeit möglich.

Thread. Der mittels `super()` an die Oberklasse übergebene Name ist optional. Er dient nur zu Debuggingzwecken. Wenn er nicht angegeben wird, dann kann `super()` wegfallen und der Name wird automatisch generiert. Die Initialisierungsanweisungen der beiden Threadobjekte werden noch streng sequentiell durchgeführt. Der Start der Ausführung eines neuen Threads erfolgt erst über die Aufruf von `start()`. Dies führt dazu, dass jeweils ein weiterer Ablauf erzeugt wird. Diese Abläufe können sofort mit der Ausführung von `run()` beginnen.

4.6.2 Threaderzeugung mittels Delegation

Da Java nur Einfachvererbung kennt, ist die Ableitung von der Klasse Thread nicht immer möglich. Die Ableitung lässt sich vermeiden, wenn man die Threadausführung an ein Objekt delegiert, das die Schnittstelle `Runnable` implementiert.

```
public class MyRunnable implements Runnable {
    private int instanzVariable;

    public MyRunnable(int parameter) {
        instanzVariable = parameter;
    }

    public void run() {
        // Hier stehen meine Anweisungen fuer den Thread-
           Ablauf.
        weitereMethode();
    }

    public void weitereMethode() {
        // Tue was.
    }

    public static void main(String[] args) {
        Thread t1 = new MyThread("Thread-A", new MyRunnable
            (7));
        MyRunnable r = new MyRunnable(8);
        Thread t2 = new MyThread(new MyRunnable(8));
        t1.start();
        t2.start();
    }
}
```

Grundsätzlich gelten hier alle Anmerkungen der ersten Variante. Es besteht aber ein deutlich erkennbarer Unterschied zwischen dem Objekten von `MyRunnable` und den Thread-Objekten.

Threadobjekte dienen der im engeren Sinn der Verwaltung eines Threadablaufs. Durch Überschreiben von `run()` kann zusätzliches Verhalten hinzu kommen. Soweit man den Thread als eigenständiges Objekt betrachtet, ist der Ablauf und seine Eigenschaften (Name, Priorität, Unterbrechungszustand usw.) gemeint. Im Programm selbst hat man nur wenig mit Thread-Objekten zu tun.

Objekte von Klassen die `Runnable` implementieren können dazu verwendet werden, mittels ihrer Methode `run()` den Anfangspunkt eines Threadablaufs anzugeben. Darin liegt aber genausowenig eine große Besonderheit, wie in einem sequentiellen Programm in der Klassenfunktion `main()`.

Aus dem Gesagten folgt, dass verschiedene Threads ohne weiteres gleichzeitig Methoden desselben Objekts ausführen können. Es ist dabei nicht nötig, dass die gemeinsam genutzten Objekte in irgendeiner offensichtlich erkennbaren Weise auf Threads Bezug nehmen.

4.6.3 Beenden von Threads und Ende des Programms

Ein Thread endet mit dem Ende von `run()`, wenn mit `System.exit()` das Ende der Programmausführung erzwungen wird, oder wenn er vor seinem Start per `setDemon(true);` zu einem „Dämonen“ erklärt wurde und alle anderen Threads, die keine „Dämonen“ sind, beendet sind. Ältere Methoden, mit denen man Threads „abwürgen“ konnte, wie `stop()`, sind wegen ihres fehleranfälligen Verhaltens *deprecated* und dürfen nicht verwendet werden.

Die Dämon-Eigenschaft ist sinnvoll für solche Threads, die im Hintergrund nützliche Hilfsaufgaben durchführen. Dies kann eine Aufgabe, wie die automatische Speicherbereinigung sein. Sequentielle Programme sind eine einfache Anwendung der angesprochenen Regel. Unbemerkt vom Programmierer verrichten dabei häufig „Dämonen“ im Hintergrund ihre Arbeit.[7] Der main-Thread, in dem das eigentliche Programm ausgeführt wird, ist der einzige Nicht-Dämon. Wenn sein Ende erreicht ist, wird auch das Programm beendet.

Bei graphischen Anwendungen (AWT oder Swing) gibt es einen besonderen Thread zur Kontrolle der gesamten Benutzerinteraktion, dem *Event-Thread*. Dieser Thread ist kein Dämon und damit beendet das Ende von `main()` auch nicht das Programm.

Es kann wichtig sein, dass man auf das Ende eines Threads wartet um über seine Ergebnisse verfügen zu können. Die einfachste Möglichkeit bietet die Methode `join()` des Threadobjekts. Es ist sichergestellt, dass nach `join()` alle durch den beendeten Thread durchgeführten Speicheränderungen sichtbar sind.

Es ist zu beachten, dass `join()` die geprüfte Ausnahme `InterruptedException` werfen kann. Dazu wird weiter unten mehr ausgeführt.

[7] Die Kölner Variante von Dämonen heißt *Heinzelmännchen*.

Kapitel 5

Das Actor-Konzept in Scala

Das Konzept der Nebenläufigkeit von Java hat große Vorteile. Die Implementierung ist sehr einfach und sie hat nur geringe Laufzeit- und Speicherkosten. Das Threading-Konzept mit gemeinsamen Speicher hat aber auch gravierende Nachteile:

- Das Threading-Modell von Java ist nicht objektorientiert. Es ist hervorgegangen aus der prozeduralen Programmierung.
- Die Kommunikation über gemeinsamen Speicher bringt das Problem der *Wettlaufbedingungen* mit sich.
- Der Schutz kritischer Bereiche fügt einem Programm die Gefahr von Verklemmungen (deadlock) hinzu.
- Fehler in nebenläufigen Programmen mit gemeinsamem Speicher lassen sich kaum reproduzieren. Die Fehler lassen sich durch Tests kaum entdecken.
- Vielen Entwicklern fällt es sehr schwer, threadsichere Programme zu schreiben. Sie neigen dazu, Multithreading zu vermeiden.

Bereits in den 70er Jahren ist im Umfeld der funktionalen Programmierung ein alternatives Modell, nämlich das Actor-Modell entstanden. Es geht zurück auf Ideen von Carl Hewitt und anderen aus dem Jahre 1973.

Im Folgenden wird das Actor-Modell in der in Scala implementierten Form beschrieben.

5.1 Das Actor-Modell

In seiner reinen Form ist das Actor-Modell ein in sich geschlossenes Konzept für die nebenläufige Berechnung im Rahmen der funktionalen Programmierung. Es passt damit zunächst nicht ganz zu den Konzepten der Objektorientierung. In Scala selbst und auch in der folgenden Darstellung sind ein paar Anpassungen vorgenommen, die hier eine Brücke schlagen.

Aus der Sicht des Actor-Modells kennt die Objektorientierung nur passive Objekte, die von einem ihnen fremden Ausführungsfaden zeitweise ins Leben gerufen werden. Das Actor-Modell stellt dagegen so etwas wie ein aktives (Funktions-) Objekt dar.

Definition:

*Ein **Actor** ist eine in sich abgeschlossene Einheit von Ablauf und von Daten. Ein Actor kann anderen Actoren Nachrichten senden, kann weitere Actoren erzeugen und kann sein Verhalten von Nachrichten abhängig machen, die er von anderen Actoren empfängt.*

Wie Sie sehen, entspricht ein Actor ziemlich genau dem, was man oft vereinfachend als die Aufgabe eines Objekts ansieht. Der entscheidende Unterschied zur „normalen“ Objektorientierung ist, dass ein Actor stets selbst entscheidet was er tut. Actoren haben sowohl formale Untersuchungen über Nebenläufigkeit als auch praktische Anwendungen beeinflusst.

Im Unterschied zu Objekten werden die Aktionen von Actoren nur durch Botschaftsobjekte und nicht durch Methodenaufrufe vermittelt. Dies ist einfach ein Ergebnis der historischen Entwicklung. Ich werde weiter unten zeigen, wie man diesen Unterschied durch das Muster der *aktiven Objekte* verwischen kann.

5.2 Nebenläufigkeit in Scala

Vorweg ist zu sagen, dass Scala das gesamte Nebenläufigkeitsmodell von Java gleichermaßen unterstützt. Von besonderem Interesse sind hier natürlich nur die mit dem Actor-Modell verbundenen Eigenschaften.

Die das Actor-Konzept unterstützenden Systemeigenschaften finden sich in erster Linie in dem Paket `scala.actors`.

Ein Actor ist in Scala eine Klasse die von `scala.actors.Actor` abgeleitet wurde. Dies bedeutetg, dass der Actor die parameterlose Prozedur `act` implementieren muss. Ein Actor wird aktiviert (gestartet) durch den Aufruf der Methode `start`. Der Lebenszyklus eines Actors endet mit dem Erreichen des Endes von `act`.

Scala-Actoren sind gleichzeitig Java-Threads. Der Unterschied besteht darin, dass ein Actor zusätzlich über eine Mailbox verfügt. Ein Actor kann Nachrichten senden und empfangen.

5.2.1 Erzeugen und Starten eines Actors

Das folgende Beispiel zeigt das Erzeugen und Starten eines Actors. Es macht (natürlich) keinen Unterschied, ob Actoren durch Klassen oder (wie in dem folgenden Beispiel) direkt durch ein Objekt implementiert sind.

```
import scala.actors._

object VeryBusy extends Actor {
  def act() {
    while (true) println("hello")
  }
}

object Main {
  def main(args: Array[String]) {
    VeryBusy.start()
  }
}
```

5.2.2 Actorerzeugung mittels der Funktion `actor`

Da man, wie Sie wissen, in Scala leicht Kontrollabstraktionen formulieren kann, indem man ganze Code-Blöcke an eine Funktion übergibt, überrascht es nicht, dass die Scala-Bibliothek eine entsprechende Funktion bereitstellt. Die Funktion `Actor.actor` startet den übergebenen Block als Actor und gibt die Referenz auf diesen Actor zurück.

Das letzte Beispiel könnte auch so aussehen:

```
import scala.actors.Actor._

object Main {
  def main(args: Array[String]) {
    val veryBusy = actor {
      while (true) println("hello")
    }
    println(veryBusy.getState)
  }
}
```

Hier muss der Actor nicht mehr gesondert gestartet werden. Die Referenz wird aber benötigt, um Methoden aufzurufen oder um Nachrichten an den Actor zu senden.

5.2.3 Datenaustausch

Bis hierher haben wir reine Nebenläufigkeit ohne Datenaustausch. Zunächst einmal müssen wir eine Einschränkung vornehmen.

Merksatz:

> *Im Actor-Modell ist es nicht zulässig, auf die Inhalte eines fremden Actors direkt zuzugreifen. Es ist auch nicht zulässig, dies mittels Aufruf einer Actor-Methode zu tun.*[1]

Die einzige erlaubte Interaktion zwischen Actoren ist das Senden und Empfangen von Nachrichten.

Grundsätzlich können beliebige Objekte (auch Zahlen) gesendet werden. Um die Sicherheit des Modells nicht zu gefährden, sollten Nachrichtenobjekte grundsätzlich unveränderlich sein. Häufig verpackt man Nachrichten in besondere Nachrichtenobjekte (`case class`). So ist es mit dem Mustererkennungsmechanismus von Scala sehr einfach, die erwartete Nachricht aus dem Briefkasten herauszufiltern.

Bei der Besprechung der Kommunikation sind verschiedene Varianten zu unterscheiden:

Asynchrone Kommunikation. Asynchron bedeutet *ohne zeitliche Kopplung*. Im Kontext der Kommunikation ist gemeint, dass das Versenden einer Nachricht niemals blockiert. Nachrichten werden sofort ausgeliefert und landen in einem *Briefkasten* (mailbox).

Synchrone Kommunikation. Synchron bedeutet *zeitlich abgestimmt*. Bei der Kommunikation bedeutet das, dass auch das Senden erst dann abgeschlossen ist, wenn auch der Empfang – in der Regel mit Rückmeldung – beendet ist.

[1] Die Actoren des ursprünglichen Konzepts sind Funktionen und können somit nur über Botschaften beeinflusst werden.

5.3 Asynchrone Kommunikation

Asynchrone Kommunikation ist in Scala die als Regelfall verwendete Kommunikationsform. Sie wird durch die Operationen `!` (ausgesprochen *send*) und `receive` ausgedrückt.

Dem Versenden von Nachrichten dient der Operator `!`. Der Sendeausdruck beginnt mit der Referenz eines Actors, gefolgt von `!` und schließlich gefolgt von dem zu versendenden Objekt. Innerhalb des versendenden Actors/Objekt hinterlässt das Versenden keine Wirkung. Senden blockiert auch nicht die Ausführung eines Actors.

```
actor1 ! "Hallo"
actor2 ! 42
actor3 ! new Rational(1,2)
```

Das Gegenstück zum Senden ist das Empfangen. Es wird ausgedrückt durch einen Receive-Ausdruck. Ein Receive-Ausdruck besteht aus `receive` gefolgt von einer partiellen Funktion.[2]

Die Bearbeitung von Receive geht so vor, dass alle in der Mailbox des Actors vorliegenden Nachrichten daraufhin überprüft werden, ob die partielle Funktion für sie definiert ist. Ist dies der Fall, wird die passende Case-Klausel ausgeführt. Ist dagegen die Funktion für keine Nachricht definiert, d.h. trifft keine der Case-Klauseln zu), dann blockiert der Actor solange bis eine passende Nachricht eintrifft. Nachrichten, die verarbeitet werden, werden aus der Mailbox entfernt.

Das folgende Beispiel zeigt ein System von Actoren. Einer davon dient als sicherer globaler Zähler.[3]

```
import actors._

case class Increment(i: Int)
case class Result(i: Int)
case object GetCount

class Counter extends Actor {
  private var count = 0

  def act() {
    while (true) receive {
      case Increment(i) => count += i
      case GetCount => sender ! Result(count)
    }
  }
}

class IncrementCounter(val cntr: Counter) extends Actor {
  def act() {
    for (i < 1 to 100) cntr ! Increment(i)
    cntr ! GetCount
    receive {
      case Result(cnt): Int => println("count = " + cnt)
    }
```

[2] `receive` ist durch eine Methode realisiert, die als Übergabe eine partielle Funktion erhällt. Sie erinnern sich? Eine partielle Funktion ist durch eine Folge von Case-Klauseln beschrieben.

[3] Case-Klassen stellen eine vereinfachte Form der Klassendefinition in Scala dar. Sie sind besonders im Kontext des Pattern-Matching von Interesse.

```
  }
}

object Main {
  def main(args: Array[String]) {
    val c = new Counter()
    c.start()
    new IncrementCounter(c).start()
    new IncrementCounter(c).start()
  }
}
```

Im Unterschied zu Java taucht hier kein `synchronized` auf. Wir haben zwar gemeinsame Objekte, aber wir haben keine gemeinsamen veränderlichen Variablen. Die einzige Veränderung, es handelt sich dabei um die Variable `count` aus der Klasse `Counter`, wird von dem Actor selbst vorgenommen.

Die Klassen `Increment`, `Result` und das Objekt `GetCount` dienen der Lesbarkeit. Etwas sparsamer hätten wir auch mit `String` und `Int` auskommen können:

```
class Counter extends Actor { // "Sparversion"
  private var count = 0

  def act() {
    while (true) receive {
      case i:Int => count += i
      case "GetCount" => sender ! count
    }
  }
 }
```

Solange man sich an die Regel hält, dass man keine Variable aus mehreren Threads verändert, gibt es keine weiteren festen Regeln, wie man die Kommunikation gestaltet. In der Praxis haben sich bisher nur ansatzweise einige Muster herausgebildet.

Es ist aber wichtig, auf einen weiteren Punkt hinzuweisen. Die Nachricht `GetCount` erwartet eine Antwort. Diese wird hier als separate Nachricht zurückgeschickt. Das sieht kompliziert aus, hat aber auch seine Vorteile.

Dadurch dass die Anfrage `GetCount` und das Ergebnis entkoppelt sind, behält die Anfrage ihren asynchronen Charakter bei. Es muss nicht direkt auf das Ergebnis gewartet werden. Es ist auch denkbar, dass erst später das Ergebnis einer Anfrage „abgeholt“ wird.

Wenn man will, kann man dabei zusätzlich die Möglichkeit eines nicht blockierenden Receive nutzen. Das folgenden Beispiel verdeutlicht dies:

```
class IncrementCounter(val cntr: Counter) extends Actor {
  for (i < 1 to 100)
    cntr ! Increment(i)
  cntr ! GetCount
  ... // beliebige Aktionen
  receiveWithin(0) {   // Timeout von 0 msec
    case Result(cnt) => println("count = " + cnt)
    case TIMEOUT =>
  }
}
```

Wenn in der angegebenen Zeit (in Millisekunden) `receiveWithin` keine der angege-

benen Nachrichten empfängt, wird eine TIMEOUT-Nachricht „empfangen“.

5.4 Synchrone Kommunikation

Man kann argumentieren, dass synchrone Kommunikation im Widerspruch zur Nebenläufigkeit steht. Und tendentiell ist das auch so. Synchronisation führt dazu, dass nebenläufige Threads ihre Ausführung aufeinander abstimmen. Im Extremfall kann es sein, dass die Nebenläufigkeit vollständig aufgehoben wird.

Umgekehrt kann die Einschränkung der Nebenläufigkeit aber auch erwünscht sein, wenn man ein bestimmtes zeitliches Verhalten erreichen will.

Es gibt Konzepte der Nebenläufigkeit, die ausschließlich auf synchroner Kommunikation aufbauen. Diese Sprachkonzepte nutzen die Effizienzvorteile aus, die man bei synchronem Datenaustausch erreichen kann.[4] Die Nachteile der Synchronisation werden durch erhöhte Nebenläufigkeit an anderer Stelle ausgeglichen. Das bekannteste Beispiel hierfür ist das Konzept CSP (*communicating sequential processes*), das von Anthony Hoare in den 80ern vorgeschlagen wurde und dann auch in verschiedenen Programmiersprachen für die Parallelverarbeitung realisiert wurde.

In Scala, dass ja auf der JVM aufbaut, lassen sich die CSP-Ideen nicht vernünftig realisieren.[5] Synchrone Kommunikation dient in Scala ausschließlich der gewollten zeitlichen Abstimmung. Synchrone Kommunikation erhöht die Gefahr von Verklemmungen. Trotzdem ist in Einzelfällen der kontrollierte Einsatz synchroner Nachrichtenübermittlung durchaus sinnvoll.

Synchrone Kommunikation hat auf der Senderseite ein Send-Receive-Schema. Es wird ausgedrückt durch den Operartor `!?` (ausgesprochen send-receive). Um die richtige von eventuell unterschiedlichen Antworten herauszufiltern, kann wieder Pattern-Matching verwendet werden.

Auf der Empfängerseite steht das Schema Receive-Reply, ausgedrückt durch die Funktionen `receive` und `reply`.

Synchrone Kommunikation eignet sich gut für die Anwendung der Kommunikation mit sicheren Datenstrukturen. Das folgende Beispiel steht für einen nebenläufigen Stack. Eine Besonderheit ist dabei, dass die Pop-Operation bei leeren Stack keine Ausnahme wirft, sondern einfach solange wartet, bis der gewünschte Wert da ist.

```
import scala.actors._
import scala.actors.Actor._

case class Push[T](x: T)
case object Pop
case object Size

class BlockingStack[T] extends Actor {
  def act() {
    var data = List[T]()
    while (true) receive {
      case Push(x) => data = x::data
      case Pop if !data.isEmpty =>
        val result = data.head
```

[4] Bei synchroner Kommunikation kann man auf die Mailbox verzichten.

[5] Es gibt mit JCSP eine CSP-Implementierung für Java. Diese ist aber, da sie nicht gut zum Konzept von Java passt, schwerfällig und ineffizient.

```
        data = data.tail
        reply(result)
      case Size => reply(data.size)
    }
  }
}
...
val stack = new BlockingStack[String]
stack.start()
...
stack ! Push("Hallo")
stack ! Push("Welt")
 ...
val x = stack !? Pop match {
  case r: String => r
}

 // Andere Variante:
 val x = (stack !? Pop).asInstanceOf[String]
```

Die letzte Zeile des Beispiels sieht etwas komisch aus. Das liegt an der vielleicht bewusst stiefmütterlichen Behandlung der synchronen Kommunikation in Scala. Es gibt nämlich für die Rückgabe durch `reply` keine Typisierung. Der statische Typ der Rücknachricht ist einfach `Any`. Wenn man den genauen Typ kennt, kann man diesen durch den Cast `asInstanceOf` angeben. Alternativ erreicht man den gleichen Effekt durch den Match-Case-Ausdruck.

Abschließend noch ein kleiner Hinweis, der auch die potentielle Gefahr des Deadlocks ausschließen hilft.

Merksatz:

Innerhalb einer Receive-Reply Anweisungsfolge sollte nach Möglichkeit keine weitere Nachricht empfangen oder synchron gesendet werden. Die Anweisungsfolge sollte in aller Regel auch nicht viel Rechenzeit beanspruchen. Wenn man dies beachtet, sind keine großen Nachteile mit Send-Receive verbunden. Insbesondere ist dann auch kein Deadlock möglich.

5.5 Aktive Objekte und Futures

5.5.1 Aktive Objekte

Synchrone Kommunikation ist auch die Grundlage für ein Muster, das in verschiedenen Programmierumgebungen immer wieder unter unterschiedlichen Namen und unterschiedlichen Varianten auftritt.

Die folgende Übersicht zeigt einige Varianten auf:

Shared Object Die Definition stammt von Brinch-Hansen und wurde als Teil des Monitor-Konzepts eingeführt. In nicht ganz so strenger Form ist es Grundlage des Java-Monitor Konzepts.

Entfernter Methodenaufruf Entfernte Methodenaufrufe sind die Grundlage für viele Architekturen verteilter Systeme. In diesem Fall verfügt jeder Methodenaufruf über einen eigenen Thread. Für den rufenden Thread sieht es so aus, als ob er allein über das entfernte Objekte verfügt.

Aktive Objekte Es gibt unterschiedliche Modelle für aktive Objekte. Allen gemein ist, dass das Objekt über einen eigenen Thread verfügt (wie in dem Scala-Actor Modell). Nach außen verfügen die aktiven Objekte über eine Methodenschnittstelle. Die Methoden kehren nach dem Aufruf sofort zurück. Der mit den Methoden verbundene Auftrag wird von dem Objekt ausgeführt, sobald sein Thread frei ist. Rückgabewerte werden dadurch realisiert, dass der aufrufende Thread beim Methodenaufruf ein sogenanntes Future-Objekt erhält, mittels dem er später das Ergebnis des Aufrufs abholen kann.

Hier soll als Beispiel einfach nur die Umsetzung von Methodenaufrufen in das Actor-Modell dargestellt werden.[6]

```
import scala.actors._
import scala.actors.Actor._

class BlockingStack[T] {
  private case class Push[T](x: T)
  private case object Pop
  private case object Size

  val stack = actor {
    var data = List[T]()

    while (true) receive {
      case Push(x: T) => data = x::data
      case Pop if !data.isEmpty =>
        val result = data.head
        data = data.tail
        reply(result)
      case Size => reply(data.size)
    }
  }
..
  def push(x: T): Unit = stack ! Push(x).
  def pop(): (stack !? Pop).asInstanceOf[T]
  def size = (stack !? Size).asInstanceOf[Int]
}
```

Durch die Methodenverpackung wird die Verwendung der Klasse deutlich einfacher. Allerdings ist auch etwas verschleiert, dass es sich um eventuell blockierende Aufrufe handeln kann.

```
val stack = new BlockingStack[String]
...
stack.push("Hallo")
stack.push("Welt")

while (stack.size != 0) println(stack.pop())
val unsinn = stack.pop() // blockiert!
```

Häufig ist das Blockieren bei Nebenläufigkeit bewusst gewollt. Trotzdem stellt es immer eine Gefahr (deadlock) dar und es reduziert auch stets den Grad der Parallelität. Der nächste Abschnitt zeigt eine einfache Methode auf, wie man diese Nachteile vermindern kann.

[6] In dieser Form eignet sich das nur für einfache Datenstrukturen mit deren Operationen nicht viel Rechenzeit verbunden ist.

5.5.2 Future

Das gerade geschilderte Konzept der aktiven Objekte bezieht seine Attraktivität aus der Ähnlichkeit mit den bekannten Methodenaufrufen. Der Nachteil der damit verbundenen synchronen Kommunikation besteht in der Herabsetzung der Nebenläufigkeit. Um diesem Nachteil zu begegnen, wurde das Muster der *Futures* entwickelt.

Definition:

> *Ein **Future**-Objekt steht für einen Wert, dessen Berechnung noch nicht abgeschlossen ist. Es ist jederzeit möglich, nachzufragen, ob das Ergebnis vorliegt. Es existiert eine Funktion, die ein ermitteltes Ergebnis zurückgibt. Existiert das Ergebnis noch nicht, blockiert diese Funktion bis das Ergebnis da ist.*

Future-Objekte können in verschiedenem Kontext verwendet werden. Sie werden auch durch die Java-Bibliothek bereitgestellt und übernehmen auch bei der Java-Nebenläufigkeit wichtige Aufgaben. Da in Scala aber viele Verwendungsmöglichkeiten einfacher auszudrücken sind, bleiben wir hier bei der Scala-Realisierung.

Für den Umgang mit Futures benötigt man aus dem Paket `scala.actors` die Klasse `Future` und das Objekt `Futures`.

Zunächst einmal kann man ein Future verwenden um eine Berechnung im Hintergrund druchführen zu können.

Das folgende Beispiel illustriert die Verwendung:[7]

```
import scala.actors._
...
val futureResult: Future[Typ] = Futures.future {
   var result: Typ = null
   ... // lang andauernde Berechnung
   resultat
}
... // andere Aufgaben
if (futureResult.isSet) ... // liegt das Ergebnis vor?

val ergebnis: Typ = futureResult() // Ergebnis abholen
```

Im Kontext der gerade besprochenen aktiven Objekte geht es darum, dass die Send-Receive Operation nicht blockiert. Dies lässt sich ebenfalls mit Futures erreichen. Aktoren definieren dafür den Operator `!!`. Dieser dient als Alternative zu `!?`. Der Unterschied besteht darin, dass `!!` nicht blockiert und sofort ein Future-Objekt zurückgibt. Später wenn mittels `reply` das Ergebnis festlegt wird, wird dann der Future-Wert gesetzt. Zu beachten ist, dass die Auswahl der Future-Funktionsweise nicht auf seiten des (Server-) Actors erfolgt, sondern beim Absenden der Nachricht (`!!`) bestimmt wird.

Als Beispiel soll eine Klasse `FutureStack` dienen, bei der die Pop-Operation selbst bei leerem Stack nicht blockiert, da sie stets sofort ein Future-Objekt zurückgibt.

```
import scala.actors._
import scala.actors.Actor._

class FutureStack {
```

[7] Die Typangaben `Typ` und `Futures[Typ]` können auch unterbleiben. Sie dienen nur der Verdeutlichung. `Typ` ist ein ausgedachter Name für einen beliebigen Typ.

```
  private case class Push[T](x: T)
  private case object Pop
  private case object Size

  val stack = actor {
    var data = List[T]()

    while (true) receive {
      case Push(x: T) => data = x::data
      case Pop if !data.isEmpty =>
        val result = data.head
        data = data.tail
        reply(result)
      case Size => reply(data.size)
    }
  }
..
  def push(x: T) { stack ! Push(x) }

  // BESONDERHEIT: FUTURE !!
  def pop() = (stack !! Pop).asInstanceOf[Future[T]]

  def size = (stack !? Size).asInstanceOf[Int]
}
```

In dem Beispiel gibt die normalerweise blockierende Pop-Operation ein Future zurück. Damit kehrt der Aufruf. von `pop` immer sofort zurück. Der Zugriff zu dem Inhalt des Resultats wird aber eventuell blockieren.

Mit dieser Stack-Variante lassen sich interessante Anwendungen machen. Im Rahmen der Nebenläufigkeit verliert der Stack seine eigentliche Bedeutung, dass die letzten Werte zuerst entnommen werden. Die Reihenfolge des Hinzufügens und Entnehmens ist nicht mehr unbedingt vorhersehbar. Das trifft insbesondere dann zu, wenn wir, was ja normalerweise nicht geht, erst die Werte entnehmen und dann erst hinzufügen:

```
val stack = new FutureStack[String]
val erstesWort = stack.pop()
val zweitesWort = stack.pop()
stack.push("Hallo")
stack.push("Welt")
println(erstesWort())
println(zweitesWort())
```

Sieht was komisch aus? Sie können sich aber sicher Anwendungen vorstellen. Das einzige formale Erkennungszeichen der Futures ist, dass die Ergebnisse bei ihrer Verwendung Funktionsklammern tragen. Futures sind halt nicht einfach Werte sondern Funktionen, die erst dann Ergebniswerte liefern, wenn diese vorliegen.

Kapitel 6

Threadsicherheit in Java

Die strenge Befolgung des Actor-Modells gewährleistet die Threadsicherheit in einem sehr hohen Maße. Dieses Modell lässt sich grundsätzlich auch nach Java übertragen. Die Kommunikation zwischen Threads muss dann durch besondere Objekte (mailbox) realisiert werden. Allerdings kann dies durch Java nicht so elegant ausgedrückt werden, wie das in Scala aufgrund der funktionalen Eigenschaften möglich ist. Außerdem ist mit dem Botschaftenaustausch auch immer eine Performanceeinbuße verbunden. Daher ist es auf jeden Fall sinnvoll, die grundlegenden Mechanismen von Java zu betrachten.

Wenn wir ein Programm mit mehreren Threads programmieren, werden wir praktisch immer Objekte benötigen, auf die mehrere Threads zugreifen. Nur so ist nämlich ein Datenaustausch zwischen den Threads sinnvoll möglich. Wir wissen bereits, dass genau dieser gemeinsame Zugriff zu Wettlaufbedingungen und damit zu unberechenbaren Fehlern führen kann.

Es gibt aber einige Mittel um Wettlaufbedingungen zu vermeiden:

- Objekte, die nur in einem einzigen Thread angesprochen werden, führen zu keinen Fehlern. Das klingt trivial. In Wirklichkeit ist das aber ein ganz wichtiges Entwurfsprinzip. Es geht dabei darum, die Kommunikation zwischen Threads so zu kanalisieren, dass sie einfach, überschaubar und auf eine geringe Anzahl von Klassen beschränkt ist.
- Die unproblematischsten Objekte sind *unveränderliche* Objekte.
- Die Java-Bibliothek stellt Klassen bereit, deren Objekte atomar und damit threadsicher modifiziert werden.
- Variablen, die als `volatile` deklariert sind, führen zu geeigneten Sichtbarkeitsregeln. `volatile` ist dann und nur dann anwendbar, wenn nicht mehrere Variable *gleichzeitig* verändert werden müssen.
- Die Sprache und die Bibliothek bieten verschiedene Möglichkeiten um einzelnen Threads den Zugriff auf bestimmte Objekte zu verwehren. Dabei handelt es sich um Varianten der *Objektsperre* (*locking*). Die Java-Mechanismen der Sperre entsprechen dem Monitor-Konzept.
- Bibliotheken können Datenstrukturen bereitstellen über die eine sichere Kommunikation möglich ist.

Bei der Diskussion geht es um die Konstruktion threadsicherer Klassen. Nur solche Klassen können als threadsicher gelten, deren Objekte grundsätzlich ohne weitere Vorkehrungen von verschiedenen Threads angesprochen werden können. Es dürfen dabei keine zusätzlichen Bedingungen an die Verwendung der Objekte gestellt werden. Threadsicherheit ist keine Eigenschaft, die sich testen lässt! Sie ist nur durch sauberes und einfaches Design zu gewährleisten.

Die hier dargestellte Liste der Mechanismen zur Threadsicherheit ist in etwa nach zunehmenden Laufzeitkosten geordnet. Dabei ist hervorzuheben, dass die Laufzeitkosten beim Multithreading nicht das primäre Entwurfskriterium sein dürfen. Die Einfachheit und Verständlichkeit geht immer vor.

Das Sperren eines Objekts erhöht aber nicht nur einfach die Laufzeit. Es kann vorkommen, dass die Programmausführung durch eine *Verklemmung* (*deadlock*) sogar vollständig blockiert wird. Dies ist ein gutes Argument dafür, Sperren so wenig wie möglich zu verwenden.

6.1 Invarianten und sicherer Konstruktor

Die Aufgabe eines Konstruktors besteht darin, das gerade erzeugte Objekt so zu initialisieren, dass seine Variablen die durch die Klasseninvariante festgelegten Anforderungen erfüllen. Dieses Ziel ist in der Regel erst am Ende des Konstruktors erreicht.

6.1.1 Der undichte Konstruktor

Damit keine Operationen auf unvollständig initialisierten Klassen möglich sind, muss man darauf achten, dass die This-Referenz des Objekts nicht vor dem Ende der Konstruktorausführung anderen Objekten bekannt wird.

Dies kann schon bei sequentiellen Programmen ein Problem sein, dann nämlich, wenn einer vom Konstruktor aufgerufenen Methode eines anderen Objekts die This-Referenz mitgegeben wird.

Für den Fall, dass die This-Referenz vorzeitig bekannt wird, zeigt das folgende Beispiel, dass selbst Konstanten einen anderen Wert haben können als im Rest des Programms.

```
@NotThreadSafe
public class LeakingConstructor {
    private final String name;

    public LeakingConstructor(String name) {
        Registry.register(this);
        this.name = name;
    }

    public String toString() {
        return name;
    }
    ...
}

/**
 * Hier werden Objekte zentral unter ihrem
 * Namen registriert.
 */
```

```
public final class Registry {
    private Registry() {}

    private static Map<String, Object> map =
        new HashMap<String, Object>();

    public static register(Object obj) {
        map.put(obj.toString(), obj);
    }

    public static Object getObject(String name) {
        return map.get(name);
    }
}
```

Das dargestellte Szenario kommt in ähnlicher Form häufig vor. Die Referenzen der gerade erzeugten Objekte sollen in einem globalen Verzeichnis abgelegt werden, so dass sie für bestimmt Aktionen immer wieder auffindbar sind. Manchmal wird dabei nur die Objektreferenz in einer Liste gespeichert, damit das Objekt von Zustandsänderungen eines anderen Objekts unterrichtet werden kann.

In diesem Fall ist das Ziel, das Objekt global über seinen Namen ansprechen zu können. Es ist angenommen, dass sich der Name aus dem Wert von `toString()` ergibt. Es ist guter Stil, wie in der Klasse `LeakingConstructor` geschehen, diesen Namen in einer unveränderlichen Variablen (`final`) abzulegen. Damit ist garantiert, dass das Objekt seinen Namen nie ändert.

Der Fehler ist nur, dass die Registrierung erfolgt, ehe das Feld `name` vom Konstruktor belegt wurde. Bei der Registrierung wird daher der Defaultwert `null` benutzt.

Fangen Sie nicht an, die Situation dadurch zu „verbessern“, dass Sie einfach nur die Reihenfolge der beiden Anweisungen im Konstruktor vertauschen. Bei sequentiellen Programmen mag das noch gehen. Bei nebenläufigen Programmen wissen Sie nicht was geschieht. Wenn der Name in einem fremden Thread abgefragt wird, kann es sein, dass sein endgültiger Wert noch nicht publiziert wurde. Der Sprachstandard garantiert jedenfalls nichts.

Merksatz:

Die This-Referenz darf vor dem Ende des Konstruktors anderen Objekten nicht mitgeteilt werden. Wenn diese Regel beachtet wird, braucht der Konstruktor auch nicht extra gegen Wettlaufbedingungen geschützt zu werden.

6.1.2 Das Muster der faulen Initialisierung

Das bekannteste Java-Entwurfsmuster ist vermutlich das *Singleton-Muster*. Es soll sicherstellen, dass innerhalb eines Systems von einer bestimmten Klassen ein einziges Objekt existiert. Außerdem soll diese Objekt erst dann erzeugt werden, wenn es wirklich benötigt wird. Obwohl es wegen seiner Einfachheit und vermeintlichen Eleganz sehr populär ist, ist seine Nützlichkeit sehr fraglich.

Die klassische Lösung sieht so aus:

```
@NotThreadSafe
public final class BadSingleton {
    private static BadSingleton instance = null;
```

```
    private BadSingleton() {
        // das Objekt kann nur intern erzeugt werden
    }

    public static BadSingleton getInstance() {
        if (instance == null) {
            instance = new BadSingleton();
        return instance;
    }
}
```

Diese Methode ist schlecht, da es leicht vorkommen kann, dass gleichzeitig zwei Threads eine Instanz der Klasse erzeugen und damit die Idee des einmaligen Objekts aushebeln. Wenn das einmalige Objekt in jedem Fall benötigt wird, bringt die „Objekterzeugung bei Bedarf“ nichts (und wenn nicht, oft auch nicht viel). Sicherer ist in jedem Fall, die sichere, wenn auch weniger elegant aussehende Methode, das Objekt direkt zu erzeugen.

```
@ThreadSafe
public final class EarlySingleton {
    private static final EarlySingleton instance =
        new EarlySingleton();

    private EarlySingleton() {
        // das Objekt kann nur intern erzeugt werden
    }

    public static EarlySingleton getInstance() {
        return instance;
    }
}
```

Hier kann kein Fehler passieren, da das einmalige Objekt bereits beim Laden der Klasse erzeugt wird. Allerdings, wird das Objekt auch dann erzeugt, wenn es nie verlangt wird. Wenn man wirklich Wert auf die späte Objekterzeugung legt, geht das aber auch:

```
@ThreadSafe
public final class SafeAndTricky {
    private static class LazyConstruction {
        static final SafeAndTricky instance =
             new SafeAndTricky();
    }

    private SafeAndTricky() {
        ...
    }

    public static SafeAndTricky getInstance() {
        return LazyConstruction.instance;
    }
}
```

Der Trick besteht hier darin, dass das Klassenobjekt von `LazyConstruction` erst bei der ersten Benutzung der Klasse und damit bei dem ersten Aufruf von `getInstance()` erzeugt wird. Gleichzeitig ist sichergestellt, dass kein Thread einen verfrühten Zugriff auf irgendeine Variable oder irgendein Objekt bekommt.

6.2 Unveränderliche Objekte

In „Algorithmen und Programmierung“ hatte ich betont, dass es sinnvoll ist, Objekte möglichst so zu definieren, dass sich ihr Zustand nie ändert. Sie verhalten sich dann wie Werte. Da sich ihr Inhalt nie ändert, sind sie gegen viele Fehler immun.

Aus diesem Grunde gibt es auch in der Java-Bibliothek eine ganze Reihe von unveränderlichen Wertobjekten. Dazu gehören die Objekte der Klasse `String` und die Objekte der Wrapperklassen `Integer`, `Double` usw.

Unveränderliche Objekte bieten den weiteren Vorteil, dass sie automatisch threadsicher sind. Um die Threadsicherheit absolut zu garantieren, müssen alle Klassen- und Instanzvariablen `final` deklariert sein. Es versteht sich von selbst, dass die im Objekt gespeicherten Komponentenobjekte ihrerseits unveränderlich sein müssen. Solange es nur um die Threadsicherheit geht, dürfen die Komponentenobjekte selbst sogar veränderlich sein. Es genügt, wenn die Komponenten wenigstens threadsicher sind.

Merksatz:

Eine Klasse, deren Klassen- und Instanzvariablen ausnahmslos `final` *deklariert sind und deren Komponenten threadsicher sind, ist threadsicher. Eine Klasse deren Variablen* `final` *deklariert sind und deren Komponenten unveränderlich sind, ist unveränderlich. Voraussetzung ist in beiden Fällen, dass der Konstruktor korrekt beendet wurde.*

Anmerkung:

In diesem Skript kennzeichne ich Klassen, die unveränderliche Objekte erzeugen, mit der Annotation `@Immutable`.

Die folgende Klasse `Rational` ist ein gutes Beispiel für eine unveränderliche Klasse. Die Unveränderlichkeit ist mit ein Grund, warum es keinen Sinn macht, Brüche nicht schon im Konstruktor sondern erst auf Aufforderung zu kürzen. Es ist dabei ein kleiner Nachteil, dass der Java-Sprachstandard und der Compiler es nicht gestatten, die Initialisierung von final-Attributen in eine Hilfsmethode zu verlegen.

```
@Immutable
public final class Rational implements
    Comparable<Rational>, Serializable {
    /*
     * Invariante: nenner > 0 und zaehler
     *             und nenner sind gekuerzt.
     *             Der Bruch aendert sich nie.
     */
    private final int zaehler;
    private final int nenner;

    public Rational(int zaehler, int nenner) {
        if (nenner == 0)
            throw new ArithmeticException();
        if (nenner < 0) {
            zaehler = -zaehler;
            nenner = -nenner;
        }
        int g = gemeinsamerTeiler(
            Math.abs(zaehler), nenner);
        this.zaehler = zaehler / g;
```

```
        this.nenner = nenner / g;
    }

    public Rational multiply(Rational r) {
        return new Rational(zaehler * r.zaehler,
            nenner * r.nenner);
    }

    ..

}
```

Anmerkung:

Unveränderliche Objekte sind grundsätzlich sicher. Daher kann es durchaus erlaubt sein, Klassenvariablen `public` *zu deklarieren und nach außen sichtbar zu machen. Ich würde dies aber nur dann tun, wenn es sich um Objekte ohne innere Logik handelt, also um Objekte, die nur dazu dienen, ein paar Inhalte zu transportieren.*

6.3 Atomare Operationen

In der Vorlesung habe ich bei der Erläuterung von Wettlaufbedingungen das Beispiel eines Zählerobjekts dargestellt, dessen Methode zum Erhöhen des Zählerstands innerhalb der Anweisung `count++;` unterbrochen werden kann. Eine zuverlässig vorhersagbare Operation verlangt, dass während der Ausführung von `count++` kein anderer Thread die Variable `count` verändert. Im Ergebnis muss es immer so aussehen, als ob eine Operation ganz oder gar nicht ausgeführt ist.

Definition:

*Eine **atomare Operation** ist ein Befehl oder eine Folge von Befehlen deren Ausführung nicht unterbrochen werden kann. Der Befehl erscheint stets ganz ausgeführt oder gar nicht.*

Die meisten Prozessoren verfügen über geeignete ununterbrechbare Operationen mit deren Hilfe sich Atomizität erreichen lässt. Der Prototyp dieser Operationen ist die atomare Operation *test-and-set* oder *compare-and-set*. Bei diesem Befehl werden zwei Grundoperationen unteilbar verbunden. Zunächst wird überprüft, ob eine Vorbedingung vorliegt (*test*) und wenn dies der Fall ist, wird eine Zuweisung ausgeführt (*set*). Mit test-and-set lassen sich sehr viele Operationen threadsicher ausführen ohne den Programmablauf zu blockieren.

Die Grundidee von test-and-set geht optimistisch davon aus, dass Wechselwirkungen zwischen Threads kaum vorkommen. In den seltenen Fällen einer unerwünschten Wechselwirkung wird die entsprechende Aktion einfach wiederholt. Die atomare Verbindung von Testen und Verändern garantiert, dass Veränderungen nur dann stattfinden, wenn sie zulässig sind.

Atomare Operationen des Namens `compareAndSet` werden in der Java-Bibliothek von verschiedenen Klassen des Pakets `java.concurrent.atomic` bereitgestellt. Ich demonstriere ihre Anwendung an dem Beispiel einer threadsicher programmierten Zählerklasse.

```
import java.util.concurrent.atomic.AtomicInteger;

@ThreadSafe
public class TestAndSetCounter {
    private AtomicInteger count = new AtomicInteger(0):

    public void count() {
        for (;;) {
            int oldCount = count.get();
            int newCount = oldCount + 1;
            if (count.compareAndSet(oldCount, newCount))
                return;
        }
    }

    public int getCount() {
        return count.get();
    }
}
```

In der Endlosschleife der Methode `count()` wird wiederholt versucht, in der Zählervariablen `count` einen um 1 erhöhten Wert zu speichern. Wirklich ausgeführt wird die Aktion nur dann, wenn zum Zeitpunkt des Abspeicherns des neuen Wertes noch der alte Wert vorliegt. Dann kann die Methode beendet werden. Aufgrund der atomaren Ausführung des Testens und Veränderns ist sichergestellt, dass bei jedem Aufruf von `count()` der Zähler exakt um 1 erhöht wird.

Es soll nicht verschwiegen werden, dass die Klasse `AtomicInteger` bereits über eine bequemere Methode des Hochzählens verfügen. Allerdings dürfte diese intern nicht viel anders als oben angegeben implementiert sein.

```
import java.util.concurrent.atomic.AtomicInteger;

@ThreadSafe
public class AtomicIntegegerCounter {
    private AtomicInteger count = new AtomicInteger(0);

    public void count() {
        count.incrementAndGet();
    }

    public int getCount() {
        return count.get();
    }
}
```

Es gibt für andere elementare Datentypen ähnliche Klassen. Noch allgemeinere Möglichkeiten erhält man mittels der Klasse `AtomicReference`. Die Anwendung kann darin bestehen, threadsicher Modifikation an Objekten vorzunehmen. Das folgende Beispiel mit dem Zähler zeigt wieder das Prinzip.

```
import java.util.concurrent.atomic.AtomicReference;

@ThreadSafe
public class AtomicReferenceCounter {
    private AtomicReference<Integer> ref =
        new AtomicReference<Integer>();
```

```
    public AtomicReferenceCounter() {
        ref.set(0);
    }

    public void count() {
        for (;;) {
            Integer oldCount = ref.get();
            Integer newCount = oldCount + 1;
            if (ref.compareAndSet(oldCount, newCount)) return
                ;
        }
    }

    public int getCount() {
        return ref.get();
    }
}
```

Wegen der notwendigen Umwandlungen von `int` zu `Integer` und umgekehrt, die hier durch Autoboxing verborgen sind, ist das für ganze Zahlen sicher nicht die beste Lösung. Das Beispiel soll nur zeigen, wie man für beliebige Objekttypen atomare Aktionen realisieren kann.

6.4 Sichere Verwendung von einfachen Variablen

6.4.1 Das Problem

Nehmen wir ein einfach aussehendes Beispiel, das aber leider nicht sicher funktioniert.

```
@NotThreadSafe
public class Iterationsverfahren implements Runnable{
    private double loesung;
    private boolean stop = false;

    public void run() {
        setzeStartwert();
        // Fehler: evtl. endlos!
        while (!stop) {
            iterationsSchritt();
        }
    }

    private void setzeStartWert() { ... }
    private void iterationsSchritt { ... }

    public void stoppen() {
        // Fehler: bewirkt evtl. nichts.
        stop = true;
    }

    public double getLoesung() {
        // Fehler: evtl. unbestimmt.
        return loesung;
    }

    public static void main(String[] a) {
        Iterationsverfahren berechnung =
```

```
            new Iterationsverfahren();
        new Thread(berechnung, "Berechnung").start();
        try {
            Thread.sleep(10000);
        } catch (InterruptedException never) {
            // Das kommt hier nie vor.
        }
        berechnung.stoppen();
        System.out.println(berechnung.getLoesung());
    }
}
```

Das Beispiel ist länger als die dahinter stehende Idee. Es geht darum, dass die Methode `iterationsSchritt()` ein Näherungsverfahren einen Schritt weiterführt. Die Methode wird wiederholt aufgerufen, so dass man eine immer bessere Näherung erwarten kann. Anstelle eines Abbruchkriteriums wird hier verlangt, dass die Berechnung einfach nach einer vorgegebenen Zeit abgebrochen wird.

Die Lösung ist aus mehreren Gründen falsch, obwohl sie ganz unschuldig aussieht. Wir haben zwei Threads, den main-Thread und den Thread „Berechnung“. Nachdem der Berechnungsthread gestartet ist, blockiert sich der main-Thread für 10 s. In dieser Zeit kann die run-Methode ohne Störung durch andere Threads ihre Berechnung durchführen. Die Iteration wird immer weiter fortgeführt, da die Variable `stop` den Wert `false` hat. Nach abgelaufener Zeit ruft der Hauptthread `berechnung.stoppen();` auf und veranlasst damit, dass `stop` zu `true` wird. Dies wiederum führt dazu, dass die Berechnung am Ende ihres Iterationsschritts anhält. Wir erwarten, dass die Ausgabe der Lösung den letzten Wert der Näherungslösung ausgibt.

Es kann sein, dass dem so ist. Es kann aber auch anders verlaufen. Die ausgegebene Lösung kann eventuell 0 oder auch irgendein anderer Wert sein. Es kann sogar sein, dass das Programm in einer Endlosschleife verbleibt und nie aufhört. Es kann auch sein, dass ein völlig unsinniger Lösungswert ausgegeben wird.

Sie sollten inzwischen wissen, dass dies mit dem sequentiellen Speichermodell von Java zusammenhängen kann, das keine Gewähr bietet, dass ein Wert, den ein Thread verändert hat, auch in einem anderen Thread sichtbar ist.

6.4.2 Die Lösung von Sichtbarkeitsproblemen mittels `volatile`

In der Tat ist die korrekte Lösung dieses Problems relativ einfach. Es nicht einmal nötig, komplexe Sperrmechanismen zu verwenden, Wir müssen nur für zwei Dinge sorgen:

1. Variablenänderungen müssen atomar verlaufen.
2. Variablenänderungen müssen in der richtigen Reihenfolge sichtbar werden.

Der erste Punkt ist sehr wichtig! In Java ist nämlich nicht garantiert, dass das Lesen und Speichern von `long` und `double` Werten nicht unterbrochen wird. Bei diesen handelt es sich um 64 bit Werte und die Spezifikation der virtuellen Maschine verlangt nur, dass elementare 32 bit Operationen stets atomar durchgeführt werden. Man kann sich vorstellen, dass es zu völlig sinnlosem Datensalat kommt, wenn man die erste Hälfte einer Zahlendarstellung liest und nach einer Unterbrechnung, in der die Zahl verändert wurde, die zweite Hälfte einer ganz anderen Zahl übernimmt.

Der zweite Punkt ist das schon angesprochene Problem der Aktualisierung von lokalen Kopien.

Beide Probleme lassen sich mit der gleichen Maßnahme lösen. Es ist nur nötig, die betreffenden Variablen als `volatile` zu deklarieren. Dann übernimmt der Compiler mehrere Garantien:

1. Die Zugriffe auf Volatile-Variable finden atomar statt.
2. Änderungen an Volatile-Variablen werden sofort publiziert.
3. Änderungen an Volatile-Variablen erscheinen im beobachtenden Threads immer in der richtigen Reihenfolge.
4. Beim Zugriff auf eine Volatile-Variable ist auch garantiert, dass andere Variable des betreffenden Threads die aktuellen Werte haben (ab der Release Java 1.5).

```
@Threadsafe
public class Iterationsverfahren implements Runnable{
    private volatile double loesung;
    private volatile boolean stop = false;

    ... wie bisher
}
```

Java garantiert inzwischen auch, dass `loesung` nach einer Änderung von `stop` korrekt sichtbar ist, auch wenn es selbst nicht als `volatile` deklariert wurde. Allerdings ist es sicher guter Still, bei der 64 bit Variablen die Atomizität der Operation zu garantieren. Auch wenn vielleicht wegen der durch Java 5 verstärkten Regeln für `volatile` dessen Overhead zugenommen hat, ist diese deutliche und klare Programmierung sicher empfehlenswert. Auf jeden Fall werden so die Blockierungsprobleme einer Objektsperre vermieden.

6.5 Threadlokale Variable

Die Klasse `ThreadLocal` stellt Variable zur Verfügung, die für jeden Thread über eine eigene Kopie verfügen, die nur von dem jeweiligen Thread abgefragt oder modifiziert werden kann. In der Regel werden threadlokale Variable als Klassenvariable verwendet, so dass man Variable erhält, die einerseits über einen globalen Namen ansprechbar sind, andererseits aber threadspezifische Wert haben. Es versteht sich von selbst, dass threadlokale Variable keinen Wettlaufbedingungen unterliegen können.

Das folgende Beispiel zeigt alle wichtigen Eigenschaften, auch wenn es wohl keine besonders sinnvolle Anwendung ist.

```
@ThreadSafe
public class LocalCounter extends Thread {
    static final ThreadLocal<Integer> count =
        new ThreadLocal<Integer>() {
            // Anfangswert wird festgelegt.
            @Override
            protected Integer initialValue() {
                return 0;
```

```
            }
        };

    final int repetitions;

    LocalCounter(int repetitions) {
        this.repetitions = repetitions;
    }

    @Override
    public void run() {
        for (int i = 0; i < repetitions; i++)
            count.set(count.get() + 1);
        System.out.println(
            Thread.currentThread.getName() +
            ": " + count.get());
    }

    // Jeder der 3 Threads zaehlt fuer sich.
    public static void main(String[] a) {
        new LocalCounter(2000).start();
        new LocalCounter(1000).start();
        System.out.println(
            "main: " + count.get());
    }
}
```

Es sind verschiedene Dinge zu beachten:

- Die threadlokalen Objekte werden explizit erzeugt und in der Regel in einer globalen Variablen gespeichert. Mit dem Typparameter kann man den Typ der gewünschten Variablen festlegen.
- Bei der Objekterzeugung wird de facto eine abgeleitete anonyme Klasse verwendet. Es ist nämlich nötig, die Methode `initialValue()` zu überschreiben um so den Anfangswert der Variablen festzulegen.
- Die Methoden `get()` und `set()` ermöglichen es, den Wert zu erfragen oder zu verändern. Hierbei ist jetzt entscheidend, dass `ThreadLocal` für jeden Thread einen separaten Speicherplatz bereit hält. Eine Veränderung in Thread A ist also nur in Thread A sichtbar.

Das Ergebnis des Programmbeispiels könnte so aussehen:

```
Thread-0: 2000
main: 0
Thread-1: 1000
```

Es wird jeweils der Threadname und der Wert der Variablen in diesem Thread ausgegeben. Man erkennt, dass es sich wohl wirklich um drei verschiedene Variablen handeln muss. Dadurch sind das Ergebniswerten sicher bestimmt. Die Reihenfolge der Ausgabezeilen kann jedoch variieren, da sie im Programm nicht festgelegt wurde.

6.6 Monitorkonzept und Sperre

Bis jetzt habe ich Methoden der Threadsicherheit besprochen, die Threadsicherheit erreichen, ohne dass damit einer oder mehrere Threads blockiert werden und warten müssen, bis der Zugriff zu einem Objekt zugelassen wird.

Blockieren steht grundsätzlich der Idee der Nebenläufigkeit entgegen. Es reduziert die Effizienz, beseitigt Möglichkeiten der parallelen Programmausführung und kann sogar dazu führen, dass das Programm vollständig „hängen bleibt“ (Deadlock). Wenn möglich, sollte man daher Objektsperren vermeiden und zumindest ihren Anwendungsbereich beschränken. In der Entwicklungsgeschichte der Java-Bibliothek lässt sich gut verfolgen, wie sich die Entwickler immer mehr von der Verwendung von Objektsperren getrennt haben.

Ganz ohne Objektsperre ist es andererseits jedoch kaum möglich, komplexe nebenläufige Java-Programme zu schreiben.

Es gibt in Java zwei Varianten zur Nutzung der Objektsperre. Zunächst gibt es in der Sprache selbst mit dem Schlüsselwort `synchronized` realisierte Verfahren und zum andern gibt es (seit Java 5) Bibliothekslösungen, die zwar etwas komplizierter in der Anwendung, dafür aber flexibler und unter Umständen auch effizienter sind.

Beide Mechanismen sind sehr ähnlich. Da die Bibliothekslösung die einfacher verständliche und allgemeinere ist, will ich mit ihrer Besprechung beginnen.

6.6.1 Kritischer Abschnitt

Die Idee der Objektsperre geht davon aus, dass es in einem Programm Anweisungsfolgen gibt, die unbedingt atomar auszuführen sind. Bei einem Computer mit einem einzigen Prozessor könnte man dies einfach dadurch erzwingen, dass der laufende Thread nicht unterbrochen werden darf. Es sollte klar sein, dass das dies keine sehr attraktive Lösung ist. Konzeptionell kommt sie der schließlich realisierten Möglichkeit aber sehr nahe. Die Java-Sperrmechanismen lassen daher immer noch zu, dass die Ausführung eines laufenden Threads zugunsten anderer Threads unterbrochen wird. Sie verhindern nur, dass andere Threads während einer kritischen Phase auf geschützte Variablen zugreifen.

Definition:

> *Ein **kritischer Abschnitt** bezieht sich auf eine Anweisungsfolge und auf eine Menge von Variablen. Er ist dadurch definiert, dass eine threadsichere Programmausführung nur dann gewährleistet ist, wenn während der Ausführung der Anweisungsfolge kein anderer Thread auf eine der betroffenen Variablen zugreift. Es muss zusätzlich gewährleistet sein, dass beim Eintritt in den kritischen Abschnitt alle zu lesenden Variablen aktualisiert sind, und dass nach Verlassen des Bereichs alle Veränderungen an Variablen publiziert werden.*

Für die folgende Diskussion wird einmal wieder das Beispiel der Zählerklasse verwendet. Zunächst sind hier nur die kritischen Abschnitte markiert. Der Deutlichkeit halber habe ich `count++` durch eine Anweisungsfolge ersetzt.

```
@NotThreadSafe
public class CriticalCounter {
    // zu schuetzende Variable
    private int count = 0;
```

```
    public void count() {
        // Beginn kritischer Abschnitt
        int oldCount = count;
        int newCount = oldCount + 1;
        count = newCount;
        // Ende kritischer Abschnitt
    }

    public int getCount() {
        // Beginn kritischer Abschnitt
        return count;
        // Ende kritischer Abschnitt
    }
}
```

6.6.2 Objektsperre mittels `ReentrantLock`

Das Paket `java.util.concurrent.lock` stellt die Mechanismen für die Objektsperre (englisch: *lock*[1]) bereit. Zunächst ist dies das Interface `Lock`. Es definiert zwei zentrale Methoden, nämlich `lock()` zur Kennzeichnung des Eintritts in einen kritischen Bereich und `unlock()` zum Kennzeichnen des Verlassens.

Die Verwendung eines Interface legt nahe, dass es durch unterschiedliche Klassen implementiert werden kann. Hier soll die Klasse `ReentrantLock` betrachtet werden.

Das Zählerbeispiel sieht jetzt wie folgt aus:

```
import java.util.concurrent.Lock;
import java.util.concurrent.ReentrantLock;

@ThreadSafe
public class LockedCounter {
    private Lock lock = new ReentrantLock();

    @GuardedBy("lock"`)
    private int counter = 0;

    public void count() {
        lock.lock(); // Beginn kritischer Abschnitt
        try {
            int oldCount = counter;
            int newCount = oldCount + 1;
            counter = newCount;
        } finally {
            lock.unlock(); // Ende kritischer Abschnitt
        }
    }

    public int getCount() {
        lock.lock(); // Beginn kritischer Abschnitt
        try {
            return counter;
        }
        finally {
```

[1] Das deutsche Wort *Schleuse* wird auch mit *Lock* übersetzt. Der Schleusenvorgang ist auch ein „kritischer" Prozess, bei dem Schiffe von einem Wasserniveau auf ein anderes gehoben/gesenkt werden. Die Sperre besteht darin, dass für den Normalfall das Fließen des Wassers verhindert wird.

```
                lock.unlock(); // Ende kritischer Abschnitt
            }
        }
    }
```

Die Sperre ist mit einem Objekt (`lock`) verknüpft. Das Objekt mit dem gesperrt wurde, stellt sicher, dass stets nur ein einziger Thread einen mit diesem Objekt geschützten Bereich betreten dar. Man kann sich das so vorstellen, dass der erste Thread der `lock()` aufruft, einen Schlüssel erhält, der ihm erlaubt, den Bereich zu betreten. Beim Verlassen des Bereichs ruft er `unlock()` auf und gibt den Schlüssel wieder zurück. Die verwendete Klasse heißt `ReentrantLock`, d.h. zu deutsch *wiedereintrittsfähige Sperre*, weil der den Schlüssel besitzender Thread mehrere geschachtelte geschützte Bereiche eines Lock-Objekts betreten kann. Erst wenn er die gleiche Anzahl `unlock()`- wie `lock()`-Aufrufe getätigt hat, wird der Schlüssel abgegeben und der kritische Bereich für andere Threads geöffnet.

Threads, die beim Aufruf von `lock()` die Sperre nicht vorfinden, werden solange in den Wartezustand versetzt, bis die Sperre verfügbar ist. Warten mehrere Threads, ist nicht festgelegt, welcher als nächster die Sperre bekommt.[2]

Wie Sie sehen, beruht der Mechanismus auf der Einhaltung bestimmter Regeln. Insbesondere ist wichtig, dass nach jedem `lock()` ein `unlock()` aufgerufen wird. Im andern Fall würden alle mit der Sperre verbundenen kritischen Bereiche dauerhaft gesperrt bleiben. Ein besonderes Problem stellt das Werfen von Exceptions dar (aber auch Return-Anweisungen, die man evtl. „`übersehen“ hat). Hier wird der kritische Bereich an beliebiger Stelle verlassen. Um zu gewährleisten, dass in jedem Fall `unlock()` aufgerufen wird, sollte man unbedingt das oben verwendete Idiom verwenden, dass der gesamte kritische Bereich in einem Try-Block steht und dann durch die Finally-Klausel garantiert wird, dass `unlock()` aufgerufen wird. Der Aufruf von `lock()` steht vor dem Try-Block, Wenn er selbst scheitert, macht es keinen Sinn, `unlock()` aufzurufen.

Selbstredend erfüllt die Sperre die gewünschte Anforderung hinsichtlich der Sichtbarkeit der Variablenwerte. Sie hat auch die positive Eigenschaft, dass sie nicht unbedingt den gesamten Programmablauf außerhalb des aktiven Threads blockiert. Ihre Semantik legt nur fest, dass Threads, die sich auf dieselbe Sperre beziehen, blockiert werden. Kritische Bereiche, die nichts miteinander zu tun haben und daher gleichzeitig ausgeführt werden können, sollten deshalb mit verschiedenen Lock-Objekten geschützt werden. Umgekehrt müssen Variablen, die koordiniert modifiziert werden, stets durch dieselbe Sperre geschützt werden.

Es ist guter Programmierstil, bei den entsprechenden Variablen zu kommentieren, wie sie geschützt werden. Dadurch zwingt man sich auch dazu, in die richtige Richtung zu denken. Dafür verwende ich hier die Annotation `@GuardedBy`.

6.6.3 Das Monitorkonzept von Brinch-Hansen

Gegen den gerade dargestellten Mechanismus des Schutzes kritischer Abschnitte durch den Aufruf der Methoden `lock()` und `unlock()` wurde immer wieder vorgebracht, dass man damit einen unsicheren Weg gewählt hat. Wird doch die deklarative Aussage, dass es sich bei einer bestimmten Folge von Anweisungen um einen schützenswerten Bereich handelt, durch imperative Anweisungen ersetzt. Dabei kann es allzu leicht passieren

[2] Wird die Sperre mit dem Konstruktoraufruf `ReentrantLock(true)` erzeugt, so ist der Wartemechanismus „fair“, d.h. der am längsten wartende Thread wird zuerst bedient.

(wenn man z.B. nicht das Finally-Idiom beachtet), dass die Objektsperre verloren geht. Ein anderer Einwand ist, dass es sich bei kritischen Abschnitten häufig um alle Methoden eines bestimmten Objekts handelt, das gerade die Variablen enthält, die zu den kritischen Abschnitten gehören. Der Verwendung der Objektsperre über dynamische Methodenaufrufe stellt gewissermaßen das falsche Ausdrucksmittel dar. Besser wäre es, nach dieser Logik, die zu schützenden Bereiche statisch zu markieren.

Die Konsequenz daraus war, zur fordern, dass Programmiersprachen Konstrukte zur Kennzeichnung schützenswerter Objekte vorhalten sollten. Dies wurde erstmals von Brinch-Hansen vorgestellt. Er hat für dieses Konzept den Begriff *Monitor* geprägt. Da wir noch nicht alle Eigenschaften von Monitoren besprochen haben, erfolgt hier zunächst eine vorläufige Definition des Begriffs.

Definition:

Ein Monitor ist ein Konzept einer Programmiersprache zum Schutz kritischer Abschnitte. Mit einem Monitor ist eine Objektsperre und die Angabe kritischer Bereiche verbunden. In der reinen Form werden alle Methoden eines gemeinsamen Objekts durch den Monitor geschützt.

Brinch-Hansen hat sein Konzept in der Sprache `Concurrent-Pascal` vorgestellt. Ich will es hier mit einer Pseudosprache (*Concurrent-Java* ?) andeuten. Als Beispiel dient wieder der Zähler.

```
@NotJava
public shared class MonitoredCounter {
    // Variablen sind automatisch private
    // Und public Methoden automatisch
    // per Sperre geschuetzt

    @GuardedBy("this")
    int counter = 0;

    public void count() {
        int oldCount = counter;
        int newCount = oldCount + 1;
        counter = newCount;
    }

    public int getCount() {
        return counter;
    }
}
```

Der große Vorteil dieses Konzepts ist, dass er garantiert, dass der Schutz gemeinsamer Objekte sicher und lesbar erfolgt. Das Attribut `shared` im Klassenkopf reicht aus, um einerseits auszudrücken, dass verschiedene Threads auf die Objekte zugreifen können und dass diese gleichzeitig hinreichend geschützt sind.

Andererseits ist damit kaum noch Spielraum bei der Implementierung effizienter anderer Mechanismen. Schwerer wiegt meiner Meinung nach, dass dieses Konzept eine nicht zu unterschätzende Gefahr von Deadlocks mit sich bringt. Dies werden wir weiter unten als *nested-monitor* Problem kennenlernen.

Einige Kernelemente des ursprünglichen Monitorkonzepts sind aber schließlich in das Monitorkonzept von Java übernommen worden:

– Das This-Objekt verwaltet die Sperre.

– Die kritischen Bereiche sind durch Anweisungsblöcke (Methoden) geschützt.

6.6.4 Das Monitorkonzept von Java

Java kennt keine besondere Art von *shared class*. Die zu schützenden Bereiche werden separat gekennzeichnet.

In der einfachsten Form sind die kritischen Bereiche Methoden und die Sperre wird durch das This-Objekt verwaltet. Die Implementierung des Zählers sieht dann in Java so aus:

```
@ThreadSafe
public class SynchronizedCounter {

    @GuardedBy("this")
    private int counter = 0;

    public synchronized void count() {
        int oldCount = counter;
        int newCount = oldCount + 1;
        counter = newCount;
    }

    public synchronized int getCount() {
        return counter;
    }
}
```

Wenn wir dies mit dem zuvor besprochenen Lock-Modell vergleichen, können wir die Methode `count()` in Pseudocode auch so umschreiben:

```
@NotJava
public void count() {
    this.lock();
    try {
        ... Anweisungen
    }
    finally {
        this.unlock();
    }
}
```

Das geht aber so nicht, da `this` ja nicht zu der Klasse `ReentrantLock` gehört. Der entsprechende Mechanismus der Synchronized-Methode funktioniert dagegen für jedes beliebige Objekt. Dies führt zu der Erkenntnis: *Wenn in Java der Monitor mit dem This-Objekt verknüpft ist, muss grundsätzlich jedes Objekt die Rolle der Sperre übernehmen können.*

Dass dies so ist, wird an einer weiteren Syntaxform von Java deutlich. Wir können uns nämlich in Java davon lösen, dass stets die ganze Methode zu schützen ist, und uns darauf beschränken, genau die kritischen Anweisungen zu schützen. Dabei können wir anstelle von `this` jedes beliebige Objekt als Sperre verwenden. Die einzige Voraussetzung ist, dass wir die zusammengehörenden Bereiche immer durch ein und dasselbe Objekt schützen.

Wieder einmal der Zähler:

```
@ThreadSafe
public class BlockSynchronizedCounter {
    private Object lock = new Object();

    @GuardedBy("lock")
    private int counter = 0;

    public count() {
        synchronized(lock) {
            int oldCount = counter;
            int newCount = oldCount + 1;
            counter = newCount;
        }
    }

    public int getCount() {
        synchronized(lock) {
            return counter;
        }
    }
}
```

Anstelle der Variablen `lock` können Sie auch beide Male (!) `this` einsetzen. Sie können auch die beiden Formen von `synchronized` mischen, müssen dann aber beachten, dass synchronisierte Methoden immer `this` als Sperre verwenden.

Die zweite Variante sieht zugegebenermaßen komplizierter als die Synchronisation einer Methode aus, so dass man (vorausgesetzt man benutzt den Java-Monitor) in der Regel erstere bevorzugen wird. Wenn man jedoch den geschützen Bereich auf Teile einer Methode beschränken will, braucht man unbedingt die Methode der synchronisierten Code-Blöcke.

Fassen wir die bisherigen Erkenntnisse für Java zusammen:

Definition:

Ein Java Monitor bezieht sich auf einen per `synchronized` *gekennzeichneten Anweisungsblock. Wenn* `synchronized` *vor einem Methodenkopf steht, fungiert* `this` *als Monitorobjekt und verwaltet die Sperre. Steht* `synchronized` *vor einem Anweisungsblock muss das Monitorobjekt explizit angegeben werden. In beiden Fällen muss beim Betreten des Anweisungsblock die Sperre erlangt werden. Gegebenfalls muss der Thread warten. Bei Verlassen des Blocks wird die Sperre zurückgegeben. Bei Klassenmethoden tritt das Klassenobjekt an die Stelle von* `this`. `synchronized` *ist nicht Teil der Methodensignatur und steht somit auch nicht in der Deklaration abstrakter Methoden.*

Es kann nicht genug betont werden, dass die Objektsperre immer in Beziehung auf eine Menge zu schützender Variablen und Objekte steht. Es ist wichtig, dies auch im Programm selbst deutlich zu machen.

Merksatz:

Verwenden Sie die Annotation `@GuardedBy` *gerade bei der Anwendung des Java-Monitorkonzepts. In Java übersieht man leicht, wie und ob Variablen geschützt sind.*

6.7 Deadlocks

Zyklische Graphen sind dadurch definiert, dass es einen Weg gibt, der wieder zu dem Ausgangspunkt eines Weges zurückführt. Ein Knoten in einem Zyklus ist sowohl sein eingener Nachfolger wie auch sein eigener Vorgänger.

Die Verwendung des gegenseitigen Ausschlusses führt Abhängigkeiten zwischen verschiedenen Threads ein, die man bildlich durch einen Abhängigkeitsgraphen darstellen an. Solange dieser Graph keine Kreise aufweist, können wir davon ausgehen, dass es keine unlösbaren Abhängigkeitsprobleme gibt. Probleme treten aber immer dann auf, wenn zwischen den Threads kreisförmige Abhängigkeiten bestehen.

Sie kennen bestimmt das Beispiel vom Hauptmann von Köpenick. Um eine Arbeit zu bekommen, musste er polizeilich gemeldet sein. Um sich anmelden zu können, musste er eine Arbeitsstelle haben. Wenn wir die Arbeitsstelle A und die Meldung M nennen und für die Abhängigkeit $\rightarrow$ schreiben, können wir das auch ausdrücken durch $A \rightarrow M \rightarrow A$. Dieses Problem ist unlösbar, d.h. der Hauptmann von Köpenick bekommt weder die Meldung noch die Arbeit, es sei denn, er versucht, wie das ja auch passiert ist, das Problem dadurch zu lösen, dass er sich die Meldung M mit Gewalt oder Betrug verschafft und damit den Kreis auflöst.

Ein anderes praktisches Beispiel ist die Rechts-vor-Links Regel im Straßenverkehr. Wenn gleichzeitig über jeden der Zufahrtswege einer Kreuzung ein Fahrzeug kommt, erhalten wir eine zyklische Abhängigkeit, die sich innerhalb der Regel nicht auflösen lässt: theoretisch müssen alle Fahrzeuge ewig warten.

In einem Programm entstehen Abhängigkeiten dadurch, dass ein Thread erst dann weitermachen kann, wenn zuvor ein anderer Thread die Sperre zurückgegeben hat. Der wartende Thread ist in diesen Fällen von Aktionen eines anderen Thread abhängig. *Wenn* dabei kreisförmige Abhängigkeiten entstehen, sind die im Kreis beteiligten Threads für alle Zeiten blockiert. In der Regel ist dies natürlich nicht gewünscht. Wir haben es bei dieser Blockierung, man nennt sie *Verklemmung* (*deadlock*), mit einem schweren Programmfehler zu tun.

Definition:

> *Eine **Verklemmung** (**deadlock**) ist eine durch eine zyklische Abhängigkeit bewirkte Verhinderung der Programmausführung. Die Abhängigkeit zwischen verschiedenen Threads entstehen, wenn diese um gemeinsame Ressourcen (z.B. Objektsperren) konkurrieren. Während die Möglichkeit von Verklemmungen (bei nebenläufigen Programmen[3]) einen Programmierfehler ist, ist ihr tatsächliches Zustandekommen vom Zufall abhängig.*

Das Kreuzungsbeispiel macht auf ein Problem aufmerksam, das auch in einem nebenläufigen Programm auftritt: Die Verklemmung muss nicht immer auftreten, sondern hängt von einem besonders ungünstigen Ablauf ab. Verklemmung ist ein Problem, dass nicht leicht durch den Programmtest erkannt werden kann. Aus der Theoretischen Informatik wissen Sie vielleicht, dass man im Kontext von Petri-Netzen versucht, durch Analyse der durch das Programm beschriebenen möglichen Abläufe herauszufinden, ob ein bestimmtes Problem eine Verklemmungsmöglichkeit enthält.

Java-Programme sind so komplex, dass man nicht in der Lage ist, alle möglichen Abläufe zu simulieren. Das ist im Detail auch nicht notwendig, da es nur um die Programmpunkte

[3] In Betriebssystemen lassen sich Verklemmungen grundsätzlich nicht immer verhindern.

geht, in denen eine Synchronisation verschiedener Threads stattfindet. Aber auch dies ist in der Regel sehr kompliziert. Der beste Weg aus diesem Dilemma besteht darin, das Programm möglichst einfach zu strukturieren und da wo es geht, von vornherein so zu schreiben, dass zyklische Abhängigkeiten nicht auftreten können.

Betrachten wir zunächst einmal eine Situation, die mit dem gegenseitigen Ausschluss verbunden ist.

```
@NotThreadSafe
public class Klasse {

    public synchronized void methode1(Klasse obj) {
        obj.methode2(); // zwei Sperren noetig !!
    }

    public synchronized void methode2() {
        ...
    }
}
```

An der Klasse alleine kann man die Verklemmungsmöglichkeit noch nicht erkennen. Dazu müssen natürlich mindestens zwei Threads beteiligt sein. Zum Aufbau der zyklischen Abhängigkeit benötigen wir aber auch mindestens zwei Objekte von `Klasse`.

Wir nehmen also an, dass es zwei Instanzen von `Klasse` gibt und dass zwei Threads A und B über jeweils eine Variable `obj1` verfügen, die auf das erste Objekt zeigt und eine Variable `obj2`, die auf das zweite Objekt zeigt.

Es geht nun darum, in welcher Reihenfolge, die beiden Threads die folgenden beiden Anweisungen ausführen.

Thread-A:

```
...
obj1.methode1(obj2);
...
```

Thread-B:

```
...
obj2.methode1(obj1);
...
```

Es gibt Abläufe, die völlig unkritisch sind. Wenn zunächst Thread A an die Reihe kommt und ohne unterbrochen zu werden, seine Anweisungen durchführen kann, dann wird auch anschließend Thread B ohne weiteres mit der Ausführung fortfahren können. Der einzige Unterschied zwischen den beiden Threads besteht darin, dass die Methoden der beiden Objekte in unterschiedlicher Reihenfolge ausgeführt werden.

1. Thread A ruft `obj1.methode1(obj2)` auf und erhält die Sperre von `obj1`.
2. Thread A ruft `obj2.methode2()` auf und hat jetzt die Sperren von `obj1` und von `obj2`.
3. Thread B versucht `obj2.methode1(obj1)` aufzurufen, wird aber blockiert, da die Sperre vergeben ist.

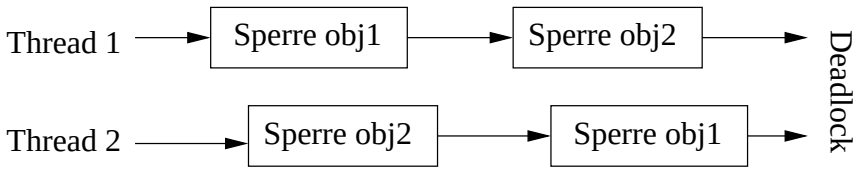

Abbildung 6.1: Ausführungsreihenfolge die zu einer Deadlock führt

4. Thread A beendet `obj2.methode2()` und gibt die Sperre von `obj2` zurück.
5. Jetzt kann der Thread B fortfahren. Er muss aber evtl. vor dem Aufruf von `obj1.methode2()` warten, bis Thread A die Sperre von `obj1` zurückgegeben hat.

Wir wissen nun nicht, in welcher Reihenfolge der Scheduler den beiden Threads Rechenzeit zuteilt, und wann er sie eventuell unterbricht um mit dem anderen Thread fortzufahren. Daher können wir auch nicht ausschließen, dass ein anderes Szenario eintritt, nämlich die in Abbildung 6.1 dargestellte Reihenfolge:

1. Thread A ruft `obj1.methode1(obj2)` auf und erhält die Sperre von `obj1`.
2. Thread B ruft `obj2.methode1(obj1)` auf und erhält die Sperre von `obj2`.
3. Thread B versucht `obj1.methode2()` aufzurufen, muss aber auf die Sperre von `obj1` warten.
4. Thread A' versucht `obj2.methode2()` aufzurufen, muss aber auf die Sperre von `obj2` warten.

Dabei ist eine Verklemmung eingetreten, da keiner der beiden Threads ausführungsbereit ist und jeder darauf wartet, dass der jeweils andere Thread eine Objektsperre zurückgibt. Diese Sperre kann aber nicht zurückgegeben werden, da der jeweilige Besitzer gerade blockiert ist, weil er auf die andere Sperre wartet.

Diese Situation entspricht übrigens genau dem Beispiel zweier dinierender Philosophen. Jeder hat vor sich einen Teller Reis. Auf dem Tisch liegen zwei Stäbchen. Zum Reisessen benötigt man nun zwei Stäbchen. Auch wenn der Tisch schlecht gedeckt ist, so ist das kein Problem, wenn beide Philosophen sich auf eine Reihenfolge einigen, also etwa zuerst A isst und dann B. Wenn sie aber beide sehr hungrig sind, kann es vorkommen, dass jeder schnell nach rechts greift (beide sind Rechtshänder) und sich damit ein Stäbchen ergattert. Wenn er aber dann kurze Zeit später nach links greift, ist das jeweilige Stäbchen schon weg. Wenn sie nun (wie echte Philosophen) beide auf ihrem grundsätzlichen gleichen Recht beharren und keiner sein Stäbchen freiwillig wieder abgibt, müssen sie elend verhungern.[4]

Was ist die Lehre aus diesem Szenario? Offensichtlich gibt es kein Problem, wenn wir nur einen einzigen Thread haben. Das ist trivial, da wir dann auch keine Abhängigkeiten

[4] Ich vermute, dass das Beispiel deshalb immer wieder nur Philosophen verhungern lässt, weil einer von ihnen, nämlich Jean Buridan, in einer berühmten Abhandlung einen Hund verhungern ließ, der gleichweit von zwei Würsten entfernt sich für keine von beiden entscheiden konnte (später hat man aus dem armen Hund einen Esel gemacht). Liebe Philosophen, Strafe muss sein!

haben können. Eine Lehre ist: „Versuche die Abhängigkeiten zwischen Threads soweit wie möglich zu minimieren.“

Es auch kein Problem, wenn jeder Thread zu jedem Zeitpunkt höchstens eine Sperre (oder nur ein Stäbchen) benötigt. Hier gibt es zwar mögliche Abhängigkeiten, es kann aber keine zyklischen Abhängigkeiten geben (zu einem Kreis, der zwei Threads umfasst, muss es wenigstens 2 Abhängigkeiten geben), auch wenn es mal vorkommen kann, dass ein Thread zeitweise warten muss.

Die Lehre aus der zweiten Situation ist: „Vermeide es, wenn möglich, dass ein Thread mehr als eine Sperre gleichzeitig besitzt!“.

Dies klingt vielleicht etwas blauäugig, ist aber in der Praxis meist leicht zu realisieren. Wir müssen nur darauf achten, dass wir vor dem Aufruf der synchronisierten Methode eines anderen Objekts die Sperre, die wir im Besitz haben, wieder freigeben. Um dies tun zu können, muss man natürlich aufpassen, dass das eigene Objekt vor der Freigabe in einem gültigen Zustand ist. Eine richtige Lösung, ohne Deadlockmöglichkeit, könnte wie folgt aussehen:

```
@ThreadSafe
public class Klasse {

    public void methode(Klasse2 obj) {
        synchronized(this) {
           ...
        }
        obj.methode2();// maximal eine Sperre !
        synchronized(this) {
            ...
        }
    }

    public synchronized void methode2() {
        ...
    }
}
```

Es kann sein, dass die Rückgabe von `obj.methode2()` in einer Instanzvariablen gespeichert werden soll, die eigentlich eines besonderen Zugriffsschutzes bedarf. Dann kann man zunächst das Ergebnis in einer lokalen Variablen festhalten und das Objekt selbst später aktualisieren:

```
private Typ instanzVariable;

public void methode(Klasse2 obj) {
    synchronized(this) {
        ...
    }
    Typ lokaleVariable = obj.methode2();
    synchronized(this) {
        instanzVariable = lokaleVariable;
        ...
    }
}
```

Auch wenn Sie vielleicht von der Theoretischen Informatik oder von der Betriebssystemvorlesung wissen, dass man in komplexen Systemen, Verklemmungen nicht (mit vernünf-

tigem Aufwand) verhindern kann, so gilt das nicht für Computerprogramme. Als Programmierer haben Sie es nämlich in der Hand, das Programm so zu entwickeln, dass es für Sie noch überschaubar bleibt. Wenn es überschaubar ist, ist es fast immer möglich, eine Lösung zu finden, in der kein Deadlock entsteht.

Kapitel 7

Kommunikationsmechanismen zwischen Threads

7.1 Warten auf Ereignisse

Bei der sequentiellen Programmierung ist es ein Fehler, wenn man aus einem leeren Behälterobjekt Daten entnehmen will. Da dies unmöglich ist, wird eine Ausnahme erzeugt (bei `ArrayList` ist es eine `IndexOutOfBoundsException`). Bei nebenläufigen Anwendungen ist die Situation aber anders. Hier kann es nämlich gewollt sein, solange zu warten, bis ein anderer Thread, die Daten geliefert hat.

7.1.1 Warten im Zusammenhang mit dem Monitorobjekt

Wir müssen zwei verschiedene Methoden für die Realisierung des Wartens haben. Der Thread, der warten will, muss dies dem System mitteilen können. Ein anderer Thread, der dafür gesorgt hat, dass ein eventuell wartender Thread wieder weiterarbeiten kann, muss dies ebenfalls dem System mitteilen. Beides wird im Monitorkonzept von Java über zwei Methoden der Klasse `Object` realisiert, nämlich durch die Methoden `wait()` und `notifyAll()` (bzw. `notify()`).

```
import java.util.LinkedList;

@ThreadSafe
public class ParallelBuffer<T> {

    @GuardedBy("this")
    private final LinkedList<T> data = new LinkedList<T>();

    /**
     * Wartet bis ein freier Platz im Puffer vorhanden ist
     * und legt ein Objekt dort ab.
     * @param x abzulegendes Objekt
     */
    public synchronized void put(T obj) {
        data.add(obj);
        this.notifyAll(); // Wartende Threads aufwecken.
    }

    /**
     * Wartet bis mindestens ein Objekt im Puffer liegt.
```

```
     * und gibt das am laengsten vorhandene Objekt zurueck.
     * @return entnommenes Objekt
     */
    public synchronized T get() throws InterruptedException {
        while (data.isEmpty())
            this.wait(); // Warten.
        return data.remove(0);
    }
}
```

Die Methode `get()` demonstriert die Logik des Wartens. Wir geben dazu eine Bedingung an, die vor der weiteren Ausführung erfüllt sein muss. Wir wiederholen das Warten solange der Puffer leer ist.

Um zu vermeiden, dass wir unnötig Rechenzeit verbrauchen, legen wir den Thread mittels `wait()` schlafen. Der Thread gibt dabei auch seine Sperre zurück.Irgendwann wird ein anderer Thread (der die freie Sperre erlangen kann), ein neues Objekt in den Puffer legen und `notifyAll()` aufrufen. Dadurch werden alle auf diesen Monitor wartenden Threads „aufgeweckt“. Sie können jedoch noch nicht sofort ausgeführt werden. Die Sperre ist ja bereits vergeben. Erst wenn die Sperre wieder frei ist, kann einer der wartenden Threads ausgeführt werden. Ein Thread darf sich nicht darauf verlassen, dass zu diesem Zeitpunkt die Bedingung, dass mindestens ein Element auf dem Stack liegt, immer noch erfüllt ist. Dies muss unbedingt erneut überprüft werden.

Die angesprochene Situation ist in der Abbildung 7.1 dargestellt. Die Abbildung macht deutlich, dass es zu jedem Objekt zwei Mengen für wartende Threads gibt. Die Threads im *Wait-Set* warten darauf, von `notify()` aufgeweckt zu werden, die Threads im *Entry-Set* warten darauf, wieder in den kritischen Bereich zu kommen. Vergleichen Sie die Zusammenhänge auch mit dem Zustandsdiagramm in Abbildung 4.4.

Effizienzüberlegungen sind nur ein Grund für die Verwendung von `wait()`. Wir befinden uns in einem geschützten Bereichs! Es ist unbedingt notwendig, während des Wartens die Sperre freizugeben, da ja sonst kein Thread in der Lage ist, etwas an dem Objekt zu ändern.

`wait()` und `notify()` sind an das Objekt der Sperre gebunden (hier `this`). Der Java-Standard schreibt vor, dass man `wait()` und `notifyALL()` nur dann benutzten darf, wenn man über die Sperre des Objekts verfügt. Das kann man sich damit erklären, dass `wait()` nur eine Objektsperre zurückgeben kann, über die der Thread verfügt. Und es ist klar dass ein `wait()` immer nur die Objektsperre seines Objekts zurückgibt und alle anderen Sperren, die der Thread besitzt weiter blockiert.! Genauso wie die zuvor beschriebene Strategie zur Vermeidung von Verklemmungen, ist dies ein Argument dafür, dass ein Thread nie mehr als eine Sperre benutzen sollte. Wenn `wait()` oder `notifyAll()` aufgerufen werden, ohne dass der Thread die Sperre besitzt, erfolgt die Auslösung einer `IllegalMonitorStateException`.

Unabhängig von den Java Sprachregeln ist eigentlich auch klar, dass `wait()` und `notifyAll()` nur in geschützten Bereiche stehen dürfen. Schließlich wird ja dabei wohl immer auf gemeinsame Variable zugegriffen.

Ich habe neben `notifyAll()` mehrfach `notify()` erwähnt. Der Grundmechanismus ist gleich. Der Unterschied ist nur, dass `notifyAll()` stets alle auf das Objekt wartenden Threads aufweckt, ein bloßes `notify()` hingegen nur einen einzigen (es ist nicht definiert welchen). Häufig wird `notifyAll()` zu viele Threads aufwecken. Wenn 10 Threads bei einem leeren Puffer warten und ein Objekt in den Puffer gelegt wird, dann kann nur ein einziger Thread weitermachen. Das ist nicht schlimm, da durch die While-

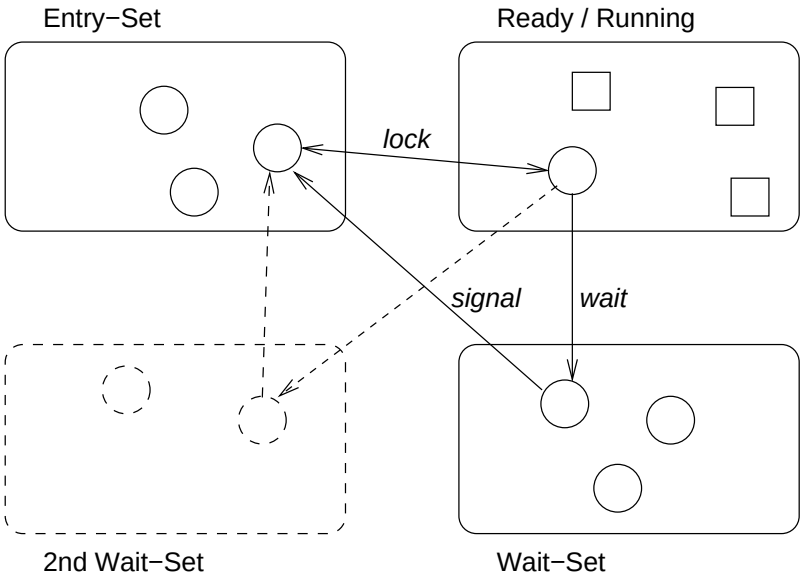

Abbildung 7.1: Mit jeder Objekt-Sperre ist ein *Entry-Set* verbunden, in dem Threads auf die Sperre warten. Zu jedem Zeitpunkt kann in der Menge der ausführbaren Threads nur einer diese Sperre besitzen (die Quadrate stellen andere Threads dar). Threads die eine Warteoperation ausführen, gelangen in den *Wait-Set* (oder in einen von mehreren). Wenn mit der Sperre ein `notify()` oder ein `signal()` aufgerufen wird, wandert einer der wartenden Threads in den Entry-Set. Bei einem `notifyAll()` oder einem `signalAll()` tun dies alle momentan im Wait-Set befindlichen Threads.

Schleife in jedem Fall die Wartebedingung erneut geprüft wird. Aber `notify()` wird sicher etwas effizienter sein.

`notifyAll()` ist jedoch trotzdem zu empfehlen, da es weniger fehleranfällig ist. Es ist nämlich besser, zu viele Threads aufzuwecken, als dass ein notwendiges Signal verloren geht.

Merksatz:

> *Solange Sie nicht absolut sicher sind, sollten Sie bei Threads nicht versuchen, zu optimieren. Wählen Sie immer die einfachste Lösung. Benutzen Sie* `notifyAll()` *und* `signalAll()`.

7.1.2 Warten im Kontext der Lock-Implementierungen

Natürlich gibt es die gleichen Möglichkeiten auch mit den Lock-Implementierungen. Hierbei erhält man wieder eine höhere Flexibilität in der Anwendung. Es ist auch möglich, mit einer Sperre mehrere Bedingungen zu verknüpfen. Dadurch kann unter Umständen nicht nur die Effizienz sondern auch die Lesbarkeit und die Programmsicherheit erhöht werden.

```
import java.util.*;
import java.util.concurrent.lock.*;

@ThreadSafe
public class ParallelBuffer<T> {
    private Lock lock = new ReentrantLock();
    private Condition notEmpty = lock.newCondition();
```

```
    @GuardedBy("lock")
    private final List<T> data = new LinkedList<T>();

    /**
     * Wartet bis ein freier Platz im Puffer vorhanden ist
     * und legt ein Objekt dort ab.
     * @param x abzulegendes Objekt
     */
    public void put(T obj) {
        lock.lock();
        try {
            data.add(obj);
            notEmpty.signalAll(); // Wartende Threads wecken.
        }
        finally {
            lock.unlock();
        }
    }

    /**
     * Wartet bis mindestens ein Objekt im Puffer liegt.
     * und gibt das am laengsten vorhandene Objekt zurueck.
     * @return entnommenes Objekt
     */
    public T get() throws InterruptedException {
       lock.lock();
       try {
           while (data.isEmpty())
               notEmpty.await(); // Warten.
           return data.remove(0);
       }
       finally {
           lock.unlock();
       }
    }
}
```

Dieser Mechanismus funktioniert genauso wie der des Java-Monitors. Die Methoden können natürlich nicht `wait` usw. heißen, da diese Namen bereits durch die Monitor-Methoden der Klasse `Object` belegt sind. Die entsprechendenMethoden heißen `signal()` und `signalAll()`.

7.2 Threadsichere Behälter

Die Java-Entwickler haben dazugelernt. In den ersten Java-Versionen sind sie bei der Anwendung des Monitor-Konzepts manchmal über das Ziel hinausgeschossen und haben die Behälterklassen `Vector`, `Hashtable` und sogar die Klasse `StringBuffer` threadsicher programmiert. Inzwischen hat man diesen Fehler korrigiert. Moderne Bibliotheksklassen sind durch die Bank nicht threadsicher. Was ist die Logik?

- In den meisten Fällen werden Behälter und andere Objekte nur in einem einzigen Thread angesprochen. Wozu soll man dann Effizienznachteile in Kauf nehmen?
- In nebenläufigen Anwendungen sind Behälter meist schon in anderen threadsicheren Objekten gekapselt. Die eigene Threadsicherheit ist dann nicht notwendig.

– Es ist einfach, Klassen per Dekoration threadsicher zu machen. Die Java-Bibliothek unterstützt dies durch Funktionen der Klasse `Collections`.

Der zweite Punkt wird z.B. an dem obigen Beispiel des `ParallelBuffer` deutlich. Ähnlich wie dort ist es nämlich häufig so, dass neben dem Zugriff auf einen Behälter noch weitere Aktionen koordiniert werden müssen.

Die folgenden Zeilen zeigen, wie man threadsichere Listen und threadsichere Maps bei Bedarf erzeugen kann.

```
import java.util.Collections;

...

List<String> list = Collections.synchronizedList(
    new LinkedList<String>();
Map<String, Object> map = Collections.synchronizedMap(
    new HashMap<String, Object>();
```

Damit sind aber noch nicht alle Probleme gelöst. Dies wird deutlich an der Iteration. Es ist nämlich nicht zulässig, während der Ausführung einer Iteration über einen Behälter (oder über eine davon abgeleitete Datenstruktur) diesen zu verändern. Tut man es doch, erfolgt ein Laufzeitfehler. Gegebenenfalls muss der Behälter durch eine Sperre – in diesem Fall unbedingt ein Java-Monitor – geschützt werden. Als Monitorobjekt ist das jeweilige Behälterobjekt zu verwenden. Das folgende Codefragment zeigt dies für die obigen Datenstrukturen `list` und `map`.

```
synchronized (list) {
    for (String s : list) { ... }
}

synchronized (map) {
    for (String k : map.keySet()) { ... }
}
```

7.3 Andere Synchronisations- und Kommunikationsmechanismen

Von Betriebssystemen, anderen Programmiersprachen und aus der Programmierpraxis sind eine Reihe anderer Mechanismen zur Verhinderung von Wettlaufbedingungen und zur Realisierung des Wartens auf bestimmte Bedingungen bekannt. Einige dieser Mechanismen sind spezialisiert auf bestimmte Anwendungsfälle, andere sind elementare Alternativen zu den Grundmechanismen von Java. Man sollte sich gelegentlich mit den verschiedenen Möglichkeiten vertraut zu machen.

Am bekanntesten ist wohl der von E. Dijkstra entwickelte Mechanismus der *Semaphore* (zu deutsch: *Ampel*, *Signal*). In der Regel ist eine (zählende) Semaphore so implementiert, dass sie über einen Zähler verfügt, der den Zugang zu einem Programmabschnitt steuert.

Die Funktion der Semaphore und des Zählers können wir uns anhand des Bildes einer Menge von Eintrittskarten veranschaulichen. Konzeptionell verfügt eine Semaphore über eine Menge von Eintrittskarten (die Anzahl wird durch den Konstruktor festgelegt).

Um eine Semaphore-Sperre zu passieren, ruft man eine Methode der Semaphore, etwa `acquire()`, auf. Wenn die Semaphore über keine Eintrittskarte mehr verfügt, wird der aufrufende Thread blockiert, andernfalls erhält er eine Eintrittskarte (der interne Zähler der Semaphore wird erniedrigt) und der Thread kann fortfahren. Wichtig ist, dass der Thread eine nicht mehr benötigte Eintrittskarte wieder zurückgibt (z.B. durch den Aufruf `release()`) und und so ermöglicht, dass ein zuvor blockierter Thread aufgeweckt wird.

Semaphore sind so allgemein verwendbar, dass sich damit sowohl die Problematik der geschützten Bereiche (anfänglicher Wert des Zählers = 1) als auch das Warten auf bestimmte Bedingungen (anfänglicher Wert = 0) realisieren lässt. Es sind aber auch andere Anfangswerte möglich. In vielen Betriebssystemen ist dies der Grund, Semaphore als Primitivkonstruktur zur Synchronisation von Prozessen zu nutzen. Dies macht jedoch in der Java-Programmierung keinen Sinn! Hier erfüllen Semaphore eine ganz andere Aufgabe, nämlich als eine höhere Abstraktion für die durch einen Zähler (Eintrittskarten) geregelte Zugangskontrolle. Die `java.util.concurrent.Semaphore` ist also nicht als Alternative zu den Mechanismen der Objektsperre zu sehen.

Neben der hier beschriebenen Realisierung einer zählenden Semaphore gibt es verschiedene andere Varianten.

Das folgende Beispiel zeigt in der Kombination von Semaphore und threadsicherem Behälter, wie sich ganz vernünftig ein beschränkter Puffer, d.h. ein Puffer, der nur eine Maximalzahl von Elementen vorhalten kann, realisiert werden kann.

```
import java.util.*;
import java.util.concurrent.Semaphore;

@ThreadSafe
public class BoundedBuffer<T> {

    private final List<T> data =
        Collections.synchronizedList(new LinkedList<T>());
    private final Semaphore available;
    private final Semaphore freeSlots;

    /**
     * Erzeugt einen beschraenkten Putter
     * @param maxSize maximale Groesse des Puffers
     */
    public BoundedBuffer(int maxSize) {
        available = new Semaphore(0);
        freeSlots = new Semaphore(maxSize);
    }

    /**
     * Wartet bis ein freier Platz im Puffer vorhanden ist
     * und legt ein Objekt dort ab.
     * @param x abzulegendes Objekt
     */
    pubic void put(T x) {
        freeSlots.acquire(); // Ist noch Platz?
        data.add(0, x);
        available.release(); // Ein Inhalt ist angekommen.
    }

    /**
     * Wartet bis mindestens ein Objekt im Puffer liegt.
     * und gibt das am laengsten vorhandene Objekt zurueck.
     * @return entnommenes Objekt
```

```
     */
    public T get() {
        available.acquire();  // Inhalt vorhanden?
        T result = data.remove(0);
        freeSlots.release();  // Ein Platz wurde frei.
        return result;
    }
}
```

Trotz der ausschließlichen Deklaration ihrer Instanzvariablen als `final` ist die Klasse `BoundedBuffer` keine Klasse für unveränderliche Objekte. Schließlich können die Liste `data` und auch die beiden Semaphoren ihren Zustand ändern Es ist aber trotzdem gut, wenn klar ist, dass man die Variablen nicht extra schützen muss. Die drei Objektattribute selbst gehören ihrerseits zu threadsicheren Klassen, so dass der Nachweis der Threadsicherheit ganz einfach ist.

Man darf auch nicht denken, dass hier keine Objektsperren vorliegen. Sie sind nur in der Liste und in den Semaphoren versteckt. Die Liste darf nicht mit den Semaphoren gemeinsam synchronisiert werden. Natürlich kann es sein, dass die Methoden nach dem Aufruf von `acquire()`, der ja den Zugriff auf die Liste regeln soll, unterbrochen werden. Gemäß dem Ticketmodell der Semaphore ist aber sichergestellt, dass ein Thread auch nach einer Unterbrechung immer noch ein Objekt oder einen freien Platz in der Liste vorfindet.

In dem Beispiel lässt die Verwendung von Semaphoren die Programmlogik deutlicher hervortreten. Objektsperren für größere Programmbereiche — die leicht zu einem Deadlock führen können — sind überflüssig.

7.4 Besondere Mechanismen

Abschließend sollen einige wenige Bibliothekslösung und Muster im Umgang mit Threads kurz angedeutet werden.

7.4.1 Vereinfachung durch Bibliotheksklassen

Die Diskussion über gegenseitigen Ausschluss, Synchronisation und Verklemmung hat gezeigt, dass mit der Nebenläufigkeit einige schwer zu erkennende und noch schwerer zu testende Probleme verbunden sind. Entsprechend gibt es auch viel Literatur zu der Frage, wie spezielle Anwendungsprobleme zu lösen sind. Alle Lösungen laufen letztlich darauf hinaus, dass man versucht, Abhängigkeiten zwischen Threads zu minimieren und möglichst einfache, an Mustern orientierte, Strukturen zu verwenden.

Eine sehr häufig verwendete Lösungsidee besteht darin, dass Threads nicht unmittelbar auf die Variablen und Methoden gemeinsamer Objekte zugreifen. Stattdessen kommunizieren Threads über gemeinsame Datenpuffer. Wenn diese Datenpuffer richtig programmiert sind, hat man alle Wettlaufbedingungen ausgeschaltet. In der Praxis können Sie dafür eine Klasse nach dem Schema des parallelen Puffers verwenden.

Eine andere Vorgehensweise kann darin bestehen, die Anzahl der gemeinsam benutzten Objekte zu minimieren. Wenn man die gesamte Synchronisation auf einige wenige Klassen konzentriert, werden Abhängigkeiten zwischen verschiedenen Programmteilen reduziert und damit wird dann die Verständlichkeit und Lesbarkeit eines Programms erhöht.

Bei großen Systemen kann sich jedoch ein Nachteil hinsichtlich der Skalierbarkeit ergeben: Ein einziges zentrales Objekt kann sich leicht als Engpass erweisen, wenn dann praktisch jeder Zugriff auf dieses Objekt mit aufwändigem Warten verbunden ist.

Das bei der Gestaltung einer einfachen Serveranwendung verwendete Konzept des *parallelen Servers* veranschaulicht ein weiteres Prinzip. Dieses besteht darin, die Aufgabenverteilung in einzelne Threads so vorzunehmen, dass es so gut wie keine Abhängigkeiten gibt. Bei dem parallelen Server wird dies dadurch erreicht, dass nach dem Aufbau der Kommunikationsverbindung mit einem anfragenden Client die weitere Kommunikation mit dem Client durch einen eigenen Thread bearbeitet wird. Diese Threads können völlig unabhängig voneinander arbeiten. Nur da wo auf zentrale Datenstrukturen zurückgegriffen wird (z.B. Liste aller Chat-Teilnehmer), ist eine Synchronisation erforderlich.

Die Klasse `Timer` bietet ebenfalls die Möglichkeit, in SpezialfäÍlen ein besonders einfaches Verhalten zu erreichen. Es geht hier darum, dass einzelne Aufgabe zu festgelegten Zeiten erfolgen sollen. Dazu bietet sich Mutlithreading an. Da die einzelnen Aufgaben aber meist unabhängig sind und meist auch zu verschiedenen Zeitpunkten ausgeführt werden, bedarf es keiner meist keiner besonderen Synchronisation. Es genügt auch, so wie es die Klasse `Timer` macht, alle Aufgaben durch einen einzigen Thread auszuführen. Die einzelnen Aufgaben werden als `TimerTask` mit einer Methode `run()` implementiert. Durch die Registrierung bei dem `Timer`-Objekt, wird dann die Ausführungsstrategie festgelegt, so dass dann zu den festgelegten Zeitpunkten die gewünschten Anweisungen ausgeführt werden.

Das folgende Beispiel zeigt die Verwendung der Klasse:

```
import java.util.Timer;
import java.util.TimerTask;

public class SimpleTimer {

    public static void main(String[] args) {
        Timer timer = new Timer();
        timer.schedule(
            new TimerTask() {
                public void run() {
                    // Tue was.
                }
            },
            0, 3000);
    }
}
```

Die Methode `schedule()` erhält drei Argumente. Zunächst das Objekt mir der auszuführenden Aufgabe, dann die Angabe der Zeitdauer bis zur ersten Ausführung und schließlich die Zeitdauer bis zur nächsten Wiederholung. Zeitangaben sind in Millisekungen.

7.4.2 Threadsicherheit in Swing

Die von der graphischen Bibliothek Swing gewählte Lösung folgt dem Prinzip, den gleichzeitigen Zugriff auf gemeinsame Daten möglichst zu vermeiden. Swing verlangt, dass Änderungen an der Graphik nur innerhalb des Event-Thread erfolgen dürfen. Natürlich müssen dort die Zugriffe auf Informationen, die von anderen Threads modifiziert werden, synchronisiert werden. Die Swingbibliothek für sich gewährleistet jedoch

keinerlei Threadsicherheit. Vielen Programmierern erscheint diese einschränkende Regel zunächst etwas kompliziert. Wenn es sie nicht gäbe, hätte der Programmierer jedoch das viel größere Problem, sich mit komplizierten Synchronisations- und Kommunikationsregeln zu beschäftigen. Außerdem würden graphischen Anwendungen mit Sicherheit einen Großteil ihrer Leistung einbüßen.

In Swing wird ein weiterer Mechanismus verwendet, der sich auch in vielen anderen nebenläufigen Systemen findet. Dieser besteht, darin, die Lösung bestimmter Aufgaben auf einen späteren Zeitpunkt zu verschieben. Die statische Methode `invokeLater()` der Klasse `SwingUtilities` erlaubt es, innerhalb eines beliebigen Threads, Programmaktionen anzustoßen, die später vom Eventthread ausgeführt werden. Die hier angesprochene Lösung ist eine Variante des „Command“-Musters, nach dem man Methoden an andere Objekte übergibt, indem man in Wirklichkeit ein Objekt übergibt, dass eine bestimmte Methode enthält.

Nachfolgend ist ein kleines Anwendungsbeispiel dargestellt.

```
public void actionPerformed(ActionEvent e) {
    // Lange Aktionen in einen Thread auslagern
    // um Eventthread nicht zu blockieren
    new Thread(new Runnable() {
        public void run() {
            // Lang laufende Taetigkeit durchfuehren
            final String text = readFile();

            // GUI nur im Eventthread aktualisieren
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    textArea.setText(text);
                }
            });
        }
    });
}
```

Die Methode `actionPerformed` könnte einem Knopf in der grafischen Oberfläche gehören. Wird nun dieser Knopf gedrückt, wird die Methode `actionPerformed` im Eventthread ausgeführt. Um ein Einfrieren der GUI zu verhindern, wird die lang laufende Aktion in einem eigenständigen Thread ausgelagert und dort durchgeführt. Das Update der GUI erfolgt jedoch wieder im Eventthread.

Wie hier gut zu sehen ist, wird die Lesbarkeit des Quelltext ziemlich behindert. Seit Java 5 steht daher die Klasse `SwingWorker` zur Verfügung. Mit ihr ist es sehr leicht möglich, Aktionen in einem separaten Thread durchzuführen und Zwischen- bzw. Endausgaben an den Eventthread zu delegieren.

7.4.3 Muster zum gesteuerten Beenden von Threads

Bereits im letzten Kapitel wurde erläutert, wann ein Thread beendet wird und welche Auswirkungen dies auf die Lebensdauer eines Prozesses hat. Hier soll jetzt angedeutet werden, wie man vorgehen kann, wenn ein Thread andere Threads veranlassen soll, sich zu beenden.

Falls in einem nebenläufigen Thread längere Aktionen ausgeführt werden, soll es von dem „Hauptthread“ aus natürlich immer noch möglich sein das Programm vorzeitig zu beenden. Ebenfalls soll in modernen GUI Anwendungen dem Benutzer die Möglichkeit

gegeben werden, angestoßene Aktionen vorzeitig wieder zu beenden. Um dies zu gewährleisten, muss der nebenläufige Thread auf „Unterbrechungswünsche“ achten und diese entsprechend durchführen:

```
@ThreadSafe
public class CancelledRunnable implements Runnable {

    public void run() {
        Thread thisThread = Thread.currentThread();
        // Wiederholung solange kein Unterbrechungswunsch
        while (! thisThread.isInterrupted()) {
            // Tue was.
            // Normale Abfrage:
            if (thisThread.isInterrupted()) break;
            try {
                // Unterbrechbare Verzoegerung.
                Thread.sleep(1000);
            }
            catch (InterruptedException interrupt) {
                break;
            }
            try
                synchronized (this) {
                    // Unterbrechbares Warten.
                    while (! bedingung())
                        this.wait();
                    // Tue was.
                }
             catch (InterruptedException interrupt) {
                 // Raeume auf.
             }
        }
    }

    public void cancel() {
        thisThread.interrupt();
    }

    ...

    public void main(String[] args) {
        CancelledRunnable runnable = new CancelledRunnable();
        new Thread(runnable).start();
        try {
            Thread.sleep(5000);
         } catch (InterruptedException neverHappens) {
         }
         runnable.cancel();
     }
}
```

Das Beispiel verdeutlicht das Prinzip des Threadabbruchs in Java. Von außen wird nur eine entsprechende Zustandsvariable gesetzt (`isInterrupted()`). Der abzubrechende Thread muss dies erkennen und die erforderliche Aktion selbst durchführen. Nur so kann garantiert werden, dass keine Objekte in einem undefinierten Zustand zurückbleiben. Das könnte jedoch passieren, würde ein Thread von außen unkontrolliert beendet, wie das mit der längst verbotenen Methode `stop()` geschehen würde.

Das Beispiel zeigt unterschiedliche Möglichkeiten, wie auf den Abbruch reagiert werden kann. In der dargestellten Form wird der mit einem Threadobjekt verbundene Unterbre-

chungsstatus genutzt. Nur dieser ermöglicht die Unterbrechung des Wartens. Alternativ lässt sich die Threadbeendigung aber auch ganz regulär in die Programmlogik implementieren. Allerdings geht dies nicht gut bei einem Threadabbruch aufgrund außergewöhnlicher Programmzustände (wie zum Beispiel Timeout). Speziell dafür ist die Unterbrechungsmöglichkeit auch gedacht.

Es bleibt anzumerken, dass das Warten auf die Objektsperre nicht unterbrochen werden kann. Dahinter verbirgt sich die Auffassung, dass die Objektsperre nichts mit der Programmlogik zu tun hat und in aller Regel nur für seltenes und kurzes Schützen eines Objekts nötig ist.

Anhang A

Glossar

Actor: Ein Actor (oder Aktor) bezeichnet einen Thread, der ausschließlich über Botschaften mit anderen Threads asynchron kommuniziert. In dem Actor-Modell von Hewitt werden in einer Mailbox vorliegende Nachrichten zum Lesen ausgewählt.

Aktives Objekt: Die Idee aktiver Objekte geht davon aus, dass jedes Objekt über seinen eigenen Kontrollfluss verfügt. Die Kommunikation zwischen Threads erfolgt nicht über die Aufrufe von (passiven) Methoden sondern über den Austausch von Botschaften. Eine Variante von aktiven Objekte verfügt als Schnittstelle über Methoden, die die Aufrufparametern entgegennehmen und unmittelbar ein Future-Objekt zurückgeben. Über dieses Future Objekt kann später auf das Resultatobjekt zugegriffen werden. Dadurch ist – wie auch bei anderen Formen des Botschaftenaustauschs – der Aufruf der Methode von ihrer Ausführung entkoppelt.

Aktives Warten: Aktives Warten bezeichnet die wiederholte Nachfrage, ob ein erwartetes Ereignis eingetreten wird. Aktives Warten vergeudet nicht nur Rechenzeit sondern kann (bei gemeinsamem Speicher) auch zu Deadlocks führen, da ja während des Wartens gemeinsame Resourcen blockiert werden.

Asynchrones Senden: Das asynchrone Versenden einer Nachricht wird ohne Rücksicht auf den Zustands des Empfängerobjekts ausgeführt. Sobald die Nachricht in einem Nachrichtenkanal abgelegt ist, kann der sendende Prozess ohne zu Warten mit seiner Befehlsausführung fortfahren. Asynchrones Senden benötigt Warteschlangen (Mailbox), in denen noch nicht gelesene Nachrichten abgelegt sind.

Atomizität: Atomizität, zu deutsch *Unteilbarkeit* bezeichnet die Eigenschaft einer Operation, dass sie immer entweder als vollständig oder als gar nicht ausgeführt erscheint. Eine atomare Operation lässt sich nie in einem Zwischenzustand beobachten.

Botschaftenaustausch: Botschaftenaustausch realisiert die Kommunikation zwischen Threads oder parallelen Prozessen auschließlich durch das Versenden von unveränderlichen Nachrichtenobjekten. Dadurch werden die mit dem Zugriff auf gemeinsame Variablen verbundenen Wettlaufbedingungen vermieden. Mit dem Warten auf eine Botschaft lässt sich auch passives Warten realisieren. Das Senden der Botschaft kann asynchron oder synchron zu dem Empfängerthread erfolgen.

Deadlock: Eind Deadlock (deutsch *Verklemmung*) entsteht, wenn sich mehrere Abläufe beim Versuch Ressourcen zu erlangen so stören, dass der Programmablauf infolge zyklischer Abhängigkeiten zum Erliegen kommt. Bei der Nebenläufigkeit spricht

man von Deadlock im Zusammenhang mit dem Zugriff auf die Objektsperre. Ein Deadlock kann nur entstehen, wenn mindestens zwei Threads um mindestens zwei Ressourcen (hier Sperren) konkurrieren. In nebenläufigen Programmen lassen sich Deadlocks nur durch eine gut überschaubare Architektur vermeiden. Ganz wesentlich ist dabei der sparsame Umgang mit Objektsperren.

Future-Objekt: Eine Future-Objekt repräsentiert das Resultat einer noch nicht abgeschlossenen Berechnung. Future-Objekte ermöglichen die Realisierung von Methoden, die ihre Berechnung nebenläufig ausführen aber unmittelbar nach dem Aufruf ein Ergebnis als Future-Objekt zurückgeben. Das Future-Objekt kann Auskunft über den Status der Berechnung geben und nach abgeschlossener Berechnung das Ergebnis mitteilen.

Gemeinsamer Speicher: Gemeinsamer Speicher bezeichnet den Zugriff unterschiedlicher Abläufe auf dieselben Speicherzellen. Auf Hardwareebene ist damit der Zugriff mehrerer Prozessoren auf dieselben Speicherbausteine gemeint. Auf Softwareebene versteht man darunter den Zugriff auf gemeinsame Variable. Die Ausprägung der Kommunikation auf Softwareebene ist unabhängig von ihrer Realisierung durch die Hardware.

Interrupt: Ein Interrupt ist eine Meldung an einen Thread. Wie diese Meldung behandelt wird, ist Sache des betroffenen Threads. Meist wird ein Interrupt jedoch verwendet, dem Thread zu signalisieren, dass er seine aktuelle Arbeit beenden und und sich kontrolliert beenden soll.

Message Passing: s. *Botschaftenaustausch*.

Monitor: Ein Monitor ist ein syntaktisch in die Programmiersprache integrierter Mechanismus zum Sperren von Objekten. Das Warten auf Bedingungen ist dabei eng mit der Objektsperre verknüpft.

Monitorobjekt: Das Monitorobjekt ist das Objekt, dass die Aktionen eines Monitors koordiniert. Die Synchronisationsmechanismen der Programmiersprache beziehen sich jeweils auf ein bestimmtes Monitorobjekt. Dieses verwaltet den *Entry-Set* der auf die Sperre wartenden Threads und den *Wait-Set* (oder mehrere Wait-Sets) der auf äußere Ereignisse wartenden Threads.

Nebenläufigkeit: Man spricht von Nebenläufigkeit, wenn ein Programm mehr als einen Kontrollfluss (Thread, Befehlsablauf) enthält. Wenn mehrere Abläufe gleichzeitig stattfinden, spricht man auch von Parallelität.

Passives Warten: Beim passiven Warten wird einem Prozess bis zum Eintreffen eines erwarteten Ereignisses die Rechenzeit entzogen. Gleichzeitig wird auch eine eventuelle Blockierung durch eine Objektsperre unterbrochen.

Polling: Unter Polling versteht man die in regelmäßigen Abständen stattfindende Abfrage, ob ein erwartetes Ereignis eingetreten ist. Im Vergleich zu passivem Warten kommt Polling ohne besondere Unterstützung durch das Laufzeitsystem aus. Es hat jedoch in der Regel erhebliche Laufzeitnachteile, da ja die Reaktion auf das erwartete Ereignis um eine feste Zeitspanne verzögert wird. Polling ist die verbreiteste Form des *aktiven Wartens*.

Priorität: Die Priorität eines Prozesses oder eines Threads beeinflusst die Rechenzeitzuteilung durch das Betriebssystem (Scheduler). Java erlaubt es, einen Thread mit einer Prioritätsangabe zu versehen. Die Auswirkung der Priorität ist allerdings durch Java nicht definiert.

Prozess: Ein Prozess ist ein in der Ausführung befindliches Programm. Das Betriebssystem stellt einem Prozess die nötigen Betriebsmittel zur Verfügung (Rechenzeit, Speicher, Ein-/Ausgabe usw.). In einem Multitasking-System können mehrere Prozesse (quasi-) gleichzeitig ausgeführt werden. Ein Prozess kann mehrere Threads beeinhalten.

Shared Memory: s. *gemeinsamer Speicher*.

Scheduling: Scheduling bezeichnett den Mechanismus eines Betriebssystems, der ausführungsbereiten Threads reale Prozessoren und damit Rechenzeit zuweist und gegebenenfalls auch wieder entzieht. Scheduling ermöglicht, dass die Anzahl der ausführungsbereiten Threads größer ist als die Anzahl der Prozessoren.

Sichtbarkeit: Hierunter versteht man die Eigenschaft, dass die aktuellen Werte gemeinsamer Variable auch in dem laufenden Thread verfügbar sind. Sichtbarkeit ist gefährdet durch Optimierungsmaßnahmen von Compiler und Laufzeitsystem. Die Synchronisationsmechanismen von Java und der Java-Bibliothek gewährleisten die Sichtbarkeit der betroffenen Variablen.

Sperre: Die Sperre ist ein durch ein Monitorobjekt verwalteter Mechanismus, der zu jedem Zeitpunkt nur einem Thread den Zutritt zu den durch das Monitorobjekt verwalteten kritischen Bereichen gestattet.

Software Transactional Memory (STM): Unter Software Transactional Memory versteht man einen Kommunikationsmechanismus der die atomare Ausführung einer logisch zusammenhängenden Operation (Transaktion) bewirkt. Häufig wird STM so realisiert, dass die Operation zunächst „optimistisch“ ohne Rücksicht auf Wettlaufbedingungen ausgeführt wird. Am Ende der Operation wird überprüft, ob sie korrekt ungestört ausgeführt werden konnte. Ist dies nicht der Fall wird die Veränderung durch die Operation wieder rückgängig gemacht und die Operation wird erneut versucht.

Synchrones Senden: Synchrones Senden ist eine Kommunikationsform bei der das Senden und das Empfangen einer Nachricht zur gleichen Zeit geschieht. Dadurch werden besondere Puffermechanismen vermieden. Synchrones Senden kann auch erwünscht sein, wenn der Sender erst nach der Bearbeitung einer Nachricht fortfahren soll. Synchrones Senden beeinhaltet aber auch stets die Gefahr eines Deadlocks.

Synchronisation: Synchronisation bedeutet zu deutsch *zeitliche Abstimmung*. In erster Linie ist damit die Koordinierung des Zutritts zu kritischen Bereichen gemeint. Die Synchronisation dient dabei dazu, Wettlaufbedingungen zu verhindern. Oft wird der Begriff auch allgemeiner für jede Art zeitlicher Abstimmung, wie das Warten auf äußere Ereignisse, verwendet.

Thread: Ein Thread (deutsch *Faden*) ist ein sequentieller Ausführungsstrang innerhalb eines Prozesses. „Normale“ sequentielle Programme verfügen über einen einzigen Thread, nebenläufige Programme haben mehrere Threads. Ein Thread hat neben dem Zugriff auf die den verschiedenen Threads gemeinsamen globalen Daten (Heap) einen threadlokalen Ausführungskontext (Stack).

Wettlaufbedingung (race condition): Eine Wettlaufbedingung liegt vor, wenn das Ergebnis einer Berechnung von zufälligen Ausführungsbedingungen, wie Scheduling oder Prozessorgeschwindigkeit, abhängt.

Threadsicher: Eine Klasse ist threadsicher, wenn die Methoden ihrer Objekte auch in einer multithreading Umgebung immer korrekt ausgeführt werden. Sie dürfen weder zu Wettlaufbedingungen noch zu Deadlocks führen.

Unveränderlich: Eine Klasse beschreibt unverändliche Objekte, wenn sich der Zustand dieser Objekte niemals ändert. Formal erkennt man unveränderliche Objekte daran, dass alle ihrer Instanzvariablen konstant (`final`) sind und nur unveränderlichr Objekte referieren. Unveränderliche Objekte sind automatisch threadsicher.

volatile: Mit dem Schlüsselwort `volatile` lassen sich Instanz- und Klasssenvariablen kennzeichnen. Eine Auswirkung ist, dass alle Variablenzugriffe, insbesondere auch bei `long` und `double` Werten atomar erfolgen. Die andere Wirkung besteht darin, dass Veränderungen immer korrekt publiziert werden, so dass sie in allen tangierten Threads sichtbar sind. Seit Java 5 gewährleistet `volatile` dass die an den Variablen vorgenommen Änderungen nie im Widerspruch zu Änderungen an anderen Variablen eines Objekts erscheinen.

Warten: Ein Thread der sich im *Wartezustand* befindet, unterliegt nicht der Auswahl als auszuführender Prozess. Er kann aus dem Wartezustand nur durch ein äußeres Signal aufgeweckt werden. Dies kann ein Signal der Prozessoruhr sein (bei `sleep()` oder zeitlich limitiertem `wait()`). In vielen Fällen wird das Beenden des Wartezustandes aber auch von anderen Threads verursacht (`notify()`, `signal()`). Das Warten ist zu unterscheiden von dem Zustand *blockiert*. Im blockierten Zustand wird zwar auch keine Rechenzeit beansprucht, das Ende der Blockierung wird jedoch nicht durch ein äußeres Signal sondern durch das Freiwerden der Sperre hervorgerufen.

Wettlaufbedingung (race condition): Eine Wettlaufbedingung liegt vor, wenn das Ergebnis einer Berechnung von zufälligen Ausführungsbedingungen, wie Scheduling oder Prozessorgeschwindigkeit, abhängt.

Literaturverzeichnis

[Blo2001] Joshua Bloch, *Effective Java*
Addison-Wesley 2001
Das Buch beschreibt anhand einiger Idiome wie man die Prinzipien der Objektorientierung in Java umsetzt. Joshua Bloch ist einer der Hauptentwickler der Java-Bibliothek.

[Eck1999] Bruce Eckel. *Thinking in Java*
http://www.BruceEckel.com
Das Buch (in elektronischer Form frei erhältlich) diskutiert sehr umfassend Sprach- und Entwurfskonzepte von Java

[Ehs2012] Erich Ehses, Lutz Köhler, Petra Riemer, Horst Stenzel, Frank Victor *Systemprogrammierung in UNIX / Linux*
Vieweg+Teubner 2012
In dem Buch werden die nötigen Grundlagen von Betriebssystemen dargestellt. Nebenläufigkeit spielt in Betriebssystemen eine große Rolle. In dem Buch sind auch die über die Betriebssysteminhalte hinausgehende Fragen des Multithreading ausführlich behandelt.

[Gam1995] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides *Entwurfsmuster*
Addison-Wesley, 1995
Das Buch der *Viererbande* hat sich inzwischen zu *dem* Standardwerk für die professionelle Entwicklung von objektorientierter Software entwickelt. Es zeigt auf, dass es für die verschiedenen Bereiche der Anwendung der Objektorientierung wiederkehrende Muster der Klassen- und Objektbeziehungen gibt. Die Kenntnis dieser Muster ist extrem hilfreich bei der Entwicklung und bei dem Verständnis komplexer Software.

[Goe2006] Goetz, Bloch, Bowbeer, Lea, *Java-Concurrency in Practise*
Addison-Wesley, 2006
Das Buch enthält eine moderne Darstellung der Nebenläufigkeit in Java. Das Niveau reicht von der gut verständlichen Darstellung der Grundlagen und der Richtlinien korrekter Programmierung bis hin zu technischen Fragen der effizienten Implementierung der Mechanismen.

[Gos2005] James Gosling, Bill Joy, Guy Steel, Gilad Bracha. *The Java Language Specification. Third Edition*
Sun Microsystems 2005
Dies ist die verbindliche und größtenteils sogar gut lesbare Festlegung der Sprachregeln von Java. Der Text ist auch über die Sun-Website erhältlich.

[Lea2000] Doug Lea *Concurrent Programming in Java, Design Prinicples and Patterns*
Addison-Wesly 2000
Dieses Buch gab den Anstoss für die Java-Concurrency-Bibliothek. Neben grundlegenden Darstellungen enthält es vielfältige und sehr detaillierte Darstellungen über verschiedene Anwendungsszenarien.

[Mor2005] Ralph Morelli, Ralph Walde *Object-Oriented Problem Solving*
Prentice Hall, 2005
Das Buch umfasst die wichtigsten etwas fortgeschrittenen Programmiertechniken.

[Schn2000] U. Schneider, D. Werner.
Taschenbuch der Informatik

Hanser-Verlag 2000
Ein sehr umfassendes Nachschlagwerk über alle wichtigen Bereiche der anwendungsorientierten Informatik.