1 Java 6 Basics

In diesem Kapitel geht es um die einfachen Sprachanpassungen in Java 5 und Ergänzungen in Java 6. In diesem Zusammenhang werden auch sprachliche Elemente angesprochen, die als Basiswissen zum Verständnis der weiteren Kapitel notwendig sind. In Java 5 zählen insbesondere dazu

- *Enumerationen*
- *Import statischer Member*
- *die for-each Schleife*
- *variable Anzahl von Argumenten, kurz Varargs*
- *Autoboxing und Unboxing primitiver Typen*

Zu den grundlegenden API-Erweiterungen im Core zählen u.a. neue

- *I/O-Methoden*
- *Floating-Point Methoden*
- *Kollektionen und das Concurrent Package*

Die Versionen Java 5 und 6, auch unter den Namen *Tiger* bzw. *Mustang* bekannt, bilden an sich eine Einheit. Java 5 war der „große Wurf". Es wurden mit Generics und Annotationen maßgebliche Sprachänderungen durchgeführt, die durch Enumerationen, Autoboxing, eine neue Art der for-Schleife sowie einem mächtigen Concurrent-Package abgerundet wurden. Aber es fehlte die Zeit für die Feinarbeit! Dieses „Finishing" ist nicht unerheblich, da es praktisch die gesamte Plattform betrifft. Erst mit dem Java 6 Release im Dezember 2006 findet es einen offiziellen Abschluss. Betrachtet man allerdings die vielen JSRs, die zum Ende 2006 offen waren, werden periodische Updates zu Java 6 unvermeidlich sein. Aber auch so ist die Liste der Änderungen in Mustang lang und eindrucksvoll. Besonders erschwerend war die allgegenwärtige Forderung nach der Kompatibilität zum alten Java bis 1.4. Ob dies noch für Java 7 alias *Dolphin* gelten wird, ist noch offen.

Das Thema dieses Buchs ist ein Überblick über fortgeschrittene Java-Techniken, die den Kern bzw. Core der Sprache ausmachen. Das setzt neben soliden Grundkenntnissen voraus, dass man die Sprachanpassungen in Java 5 sowie das aktuelle Core-API von Java 6 einzusetzen versteht. Deshalb wird dieses Standardrepertoir in diesem einführenden Kapitel zuerst vorgestellt. Ein Ignorieren dieser neuen Java 5-Features ist nicht mehr möglich, denn seit der Einführung in 2004 tauchen sie in jedem neuen Code auf!

Bis auf die Enumerationen laufen alle Sprachänderungen unter dem Label „nice to have“, treffender auch mit „syntaktischer Zucker“ tituliert. Denn intern werden sie vom Compiler auf die alten Konstrukte umgesetzt. Aber das ist bereits von inneren Klassen bekannt, die auch nur der Compiler kennt. Die JVM kennt jedenfalls nur äußere Klassen.

Nachfolgend wird alles so einfach und kurz wie möglich dargestellt. Enumerationen werden beispielsweise nur in ihrer simplen Form angesprochen, ohne auf ihren generischen Hintergrund einzugehen. Generics und Annotationen werden zwar der Vollständigkeit bzw. Korrektheit halber verwendet, aber alles ohne großen Hintergrund. Die Details werden in den folgenden Kapitel ohnehin nachgeholt. Im zweiten Teil dieses Kapitels werden die API-Änderungen am Java 6 Core besprochen. Was zum Core gehört, ist nicht unbedingt einfach abzugrenzen. Deshalb beschränken wir uns auf den Teil, der von jedem Java-Programmierer verstanden werden muss. Dazu gehören sicherlich die Kollektionen. Außen vor bleibt erst einmal das Scripting- und Compiler-API. Auch XML oder JDBC würden einfach den Rahmen dieses Kapitels sprengen.

1.1 Enumerationen

Enumerationen sind den C/C++ Programmierern schon seit langem vertraut. Selbst die Lehrsprache Pascal kannte sie bereits. Bis zur Version 5 musste man Enumerationen in Java entweder als konstante statische ganze Zahlen (normalerweise vom Typ `int`) definieren oder eigenhändig Klassen bauen. Das erstere ist recht fehleranfällig, das letztere recht aufwändig und aufgrund von Problemen bei der Serialisierung sogar überaus anspruchsvoll.

Deshalb waren Enumerationen überfällig und können in ihrer einfachen Form wie in C/C++ geschrieben werden. Im Gegensatz zu anderen Konventionen gibt es keine verbindliche Schreibweise von Enumerationen. Folgende Variationen sind beliebt:

```
enum Currency { DOLLAR, EURO, YEN }
enum currency { DOLLAR, EURO, YEN }
enum Currency { dollar, euro, yen }
enum currency { dollar, euro, yen }
```

Die Präferenz liegt auf der ersten Schreibweise. Der Grund: In Wirklichkeit ist eine `enum` eine spezielle Art von Klasse und Klassennamen beginnen mit einem Großbuchstaben. Da `enum`-Member Konstante sind, werden sie laut Java-Konvention groß geschrieben. Für die Leser, die enums aus anderen Sprachen kennen, vorab ein Alleinstellungsmerkmal von Java. Wie auch in C/C++ stecken hinter den einzelnen symbolische Namen wie `DOLLAR` ganze Zahlen, aber in Java kann man sie nicht selbst setzen. Die Zahl `0` steht immer für die erste Konstante, d.h. `DOLLAR` wird intern als `0` gespeichert. Ein explizites Setzen wie nachfolgend in C/C++ ist nicht erlaubt:

```
enum Currency { DOLLAR=0, EURO, YEN }  // Fehler
```

Die Verwendung von Enumerationen ist somit auch typsicher. Man kann sie – vergleichbar den `boolean` Werten `true` und `false` – nicht als ganze Zahlen ansprechen bzw. zu `int` casten. Die folgenden Konstrukte sind alle nicht erlaubt:

```
Currency cur= 1;
if (cur==1) {      // Fehler
    ...
}
switch (cur) {
  case 0: /* ... */     // Fehler
  case 1: /* ... */     // Fehler
  case 2: /* ... */     // Fehler
}
```

Das nächste Beispiel zeigt, dass die Syntax von `switch` erweitert wurde.

```
Currency cur= Currency.EURO;

if (cur==Currency.YEN) { // ohne enum-Name cur==YEN wäre ein Fehler!
    ...
}

switch (cur) {
  case DOLLAR:  /* ... */
  case EURO:    /* ... */
  case YEN:     /* ... */   // mit enum-Name Currency.YEN wäre ein Fehler!
}
```

Im Gegensatz zu `if` darf im `switch` hinter `case` kein Präfix `Currency` verwendet werden. Will man unbedingt an die ganze Zahl hinter der `enum`-Konstante heran, gibt es dafür die Methode `ordinal()`:

```
if (cur.ordinal() == 1)
  System.out.println("EURO");
System.out.println(cur);        // so ist die Ausgabe von EURO eleganter!
```

Die Konstanten werden aufgrund einer entsprechenden `toString()` per Default mit ihrem Namen ausgegeben. Dagegen gibt es keine Methode, die zu den ganzen Zahlen die zugehörige Konstante liefert. Man muss schon exakt den symbolischen Namen als `String` schreiben:

```
cur= Currency.valueOf("YEN");
cur= Currency.valueOf("Yen");  // löst eine Ausnahme aus!
```

Die letzte Anweisung wird kompiliert, führt aber zur Laufzeit zu einer `IllegalArgumentException`. Neben `ordinal()` gibt es noch eine nützliche statische Methode `values()`, die alle Enumerations-Member liefert (siehe Beispiel in Abschnitt 1.3).

1.2 Import statischer Member

Betrachtet man im letzten Abschnitt Anweisungen wie

```
Currency cur= Currency.EURO;
if (cur==Currency.YEN)
  // ...
```

stört das Präfix `Currency` vor den Konstanten. Die Verwendung in switch-case ist eleganter. Bisher musste in Java jedem statischen Member, ob Feld oder Methode, der Klassenname vorangestellt werden, sofern man außerhalb der zugehörigen Klasse darauf zugreifen wollte. Also hat man sich bei Sun schweren Herzens zu einer *statischen* `import`-Anweisung entschlossen. Damit sehen Felder und Methoden plötzlich wie globale Variable oder Funktionen in C aus und es ist im Code nicht mehr sofort klar, aus welcher Klasse sie eigentlich stammen. Obwohl dies zu negativen Reaktionen geführt hat, ist bei diszipliniertem Einsatz statischer Imports der resultierende Code einfach schöner:

```
// --- zuerst die Imports mit Package-Angabe, hier kap01
import static java.lang.Math.*;
import static kap01.Currency.*;
import static java.lang.System.*;
// ...

// --- der Zugriff ist nun über die einfachen Namen möglich
Currency cur= EURO;
if (cur==YEN)
  // ...

// --- ohne Präfix System.
out.println(PI);
```

Die Angabe des Package beim statischen Import darf nicht fehlen, selbst wenn man sich im selben Package wie die Klasse mit den statischen Membern befindet. Das Package zu diesem Kapitel hört auf den Namen `kap01`.

1.2.1 Das Ende eines Anti-Patterns

Mit Einführung des statischen Imports wird mit einem so genannten Anti-Pattern aufgeräumt. Um den Zugriff über den einfachen Namen auf Konstante in der Vergangenheit zu ermöglichen, wurden Interfaces definiert, die nur Konstante enthielten. Die Klasse, die dann auf diese Konstante unter dem einfachen Namen zugreifen wollte, brauchte dann nur das Interface zu implementieren. Da es wie ein Marker-Interface keine Methoden enthielt, reichte ein simples `implements`:

```
// --- das Anti-Pattern in Aktion:
interface LengthUnit {
  int CM= 1;
  int  M= 2;
  int KM= 3;
}

public class Any implements LengthUnit {
  // ...
  public void foo() {
    System.out.println(KM);   // Zugriff über einfachen Namen
  }
}
```

Was ist am dem Pattern so anti? Es ist ein klassisches Beispiel für den Missbrauch eines Interfaces! Das Interface wird zur Implementierung von Konstanten benutzt. Zu was dies führen kann, zeigt diese einfache Zuweisung:

```
LengthUnit lu= new Any();
```

Die macht überhaupt keinen Sinn. Das Interface `LengthUnit` hat kein Verhalten, nur diverse Konstante. Was soll also ein Typ wie `LengthUnit` oder eine Variable wie `lu` eigentlich bedeuten? Der neue statische Klassen-Import hat nun den gleichen Effekt. Fasst man die Konstanten in einer Klasse zusammen, bewirkt ein statischer Import einen Zugriff über den einfachen Namen. Es gibt somit keinen Grund mehr für den Einsatz des Anti-Patterns, basierend auf kuriosen Interfaces.

1.3 Interfaces vs. Klassen

Für das Verständnis von Kollektionen, Generics oder einer *Service Orientierten Architektur* (SOA) ist die klare Unterscheidung zwischen Verhalten (*Behavior*) und Zustand (*State*) notwendig. Ein Interface fasst eine Verhaltensweise bzw. einen Service von Objekten unter einem gemeinsamen Namen zusammen. Da ein Interface keine Implementierung des Service enthält, fehlen die Felder, die für die Speicherung des Zustands einer Instanz benötigt werden.

Klassen spielen dagegen in erster Linie die Rolle von *Objekt-Fabriken*. Dadurch, dass sie mehrere Interfaces implementieren können, kann eine Klassen-Instanz durchaus je nach Bedarf unterschiedliche Services anbieten. Die Instanzen spricht man dann auch idealerweise über eine Referenz des gerade benötigten Interface/Service an und ist so frei von der Art der konkreten Implementierung. Umgekehrt – wenn auch nicht direkt in der Sprache unterstützt – kann man den Service eines Interfaces auf mehrere Klassen verteilen. Das wird in der Regel über ein Proxy-Objekt realisiert, das ohnehin nur als Interface benutzt werden kann. Es verteilt dann intern den Service auf mehrere Klassen.

Somit hat man eine m-n Beziehung zwischen Klassen und Interfaces. Leider wird bei Java der Begriff Typ für Klassen und Interfaces gleichermaßen benutzt. Das ist nicht schön, denn es führt bei den Begriffen Sub/Super-Typ bzw. Einfach/Mehrfach-Vererbung zu Irritationen. Denn Typ-Beziehungen von Interfaces unterliegen der Mehrfach-Vererbung, Typ-Beziehungen von Klassen der Einfach-Vererbung. Die Zusammenhänge sind in Abbildung 1.1 als UML-Diagramm dargestellt.

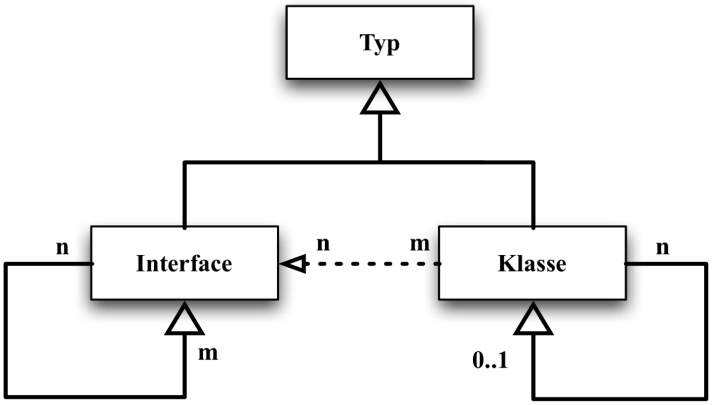

Abbildung 1.1 Beziehungen von Typ, Interface, Klasse

Eine klare begriffliche Unterscheidung wäre sicherlich schöner. Im Idealfall steht Typ für Interface. Bei Klassen ohne explizites Interface ist es die Menge der `public` deklarierten Methoden. Auf den Punkt gebracht:

Hinweis 1.1 Typen

- Ein Interface beschreibt mit seinem Namen das Verhalten bzw. den Service eines Objekts in Form von einer Menge von Methoden unabhängig von der Implementierung. Da die Methoden nur in Verbindung mit dem Typ-Namen benutzt werden können, spricht man von einem *nominalen Typsystem*.
- Subtypen spezialisieren den Service durch Hinzufügen weiterer Methoden. Deklariert als Interfaces, erlauben sie Mehrfach-Vererbung. Diese Beziehung wird mit *IS-A* bezeichnet, eine Kurzform von „ist eine besondere Art von“.
- Klassen implementieren das geforderte Verhalten und sind Fabriken für Instanzen. Die (Einfach-)Vererbung dient dazu, Felder und Code auf Subklassen zu übertragen.
- An der Spitze einer Klassen-Hierarchie findet man häufig abstrakte Klassen, die den generellen Code enthalten, der von allen konkreten Klassen genutzt werden kann. Dies vereinfacht die Implementierungen.
- Eine Klassen-Hierarchie braucht nicht mit der Sub/Super-Typ-Beziehung übereinzustimmen und kann sogar der logischen Typ-Beziehungen widersprechen.

1.3.1 Typ vs. Implementierung

Der letzte Punkt im Hinweis ist die eigentliche Schwierigkeit. Zum Beispiel sind im Collection-Framework die Container nach mathematischen Gesichtspunkten als Interfaces definiert. Bei der Implementierung findet man dagegen aus Effizienzgründen Klassen, die mehr als einen Typ implementieren. Man wäre schlecht beraten, ihre Instanzen als Klassentyp anzusprechen, da man dann Methoden aus widersprüchlichen Typen verwenden könnte.

Implementierung- und Typ-Beziehungen können sogar konträr sein. Hierzu drei Beispiele:

1. Typ-Beziehung	*2. Typ-Beziehung*
`Ellipse extends Circle`	`Circle extends Ellipse`
`Complex extends Real`	`Real extends Complex`
`Point3D extends Point2D`	`Point2D extends Point3D`

Jeder der drei Typen `Circle`, `Point2D` und `Real` hat Felder, die die zugehörigen Super-Typen `Ellipse`, `Point3D` oder `Complex` recht gut verwenden können. Für die Implementierung ist also die erste Typ-Beziehung von Vorteil. Alle Felder können wiederverwendet werden. Die zweite Typ-Beziehung ist aus der Sicht der Implementierung dagegen recht ungünstig, da man unnötige Felder inklusive der Getter und Setter vererbt. Die machen in der Subklasse keinen Sinn. Aber die erste Typ-Beziehung ist aus Typ-Sicht falsch. Die Beziehung lautet: „Kreis is-a Ellipse“ bzw. „ein Kreis ist ein besondere Art von Ellipse, bei der die Hauptachsen a und b gleich lang sind“. Setzt man sich über diese Typ-Beziehungen hinweg und wählt die erste Alternative, gibt es unerwünschte Effekte bei Methoden, die als Argumente Kreise, 2-dim Punkte oder reelle Zahlen erwarten. Diesen Methoden kann man Ellipsen, 3-dimensionale Punkte oder komplexe Zahlen übergeben, die zusätzliche Eigenschaften haben.[1] Verhindern kann man es nicht, da eine Subtyp-Instanz überall da übergeben werden kann, wo ein Supertyp erwartet wird. Fazit: Man steckt in einem Dilemma!

Retrofitting

In alten Java-Versionen wurde immer nur die konkrete `String`-Klasse als Parameter bei Methoden benutzt, die auch mit anderen string-artigen Klassen arbeiten könnten. Nachteil dieser Art werden öfter mittels *Retrofitting* beseitigt. Im `String`-Fall führte man ein Interface `CharSequence` ein, welches nachträglich von allen string-artigen Klassen implementiert wird. Nun kann man Methoden deklarieren, die als Parameter `CharSequence` verwenden und nicht auf `String` fixiert sind. Diese Methoden können gleichermaßen von den konkreten Klassen wie `String`, `StringBuffer`, `StringBuilder` und `CharBuffer` verwendet werden.

[1] Rein funktional betrachtet, kann man das eventuell als korrekte Subtyp-Beziehung ansehen!

1.4 for-each Schleife

Die neue for-each Schleife wird vom Compiler intern in die alte `for` Schleife umgewandelt, ist also nur syntaktischer Zucker. Sie ist recht praktisch, da sie den Zugriff auf die Elemente von Arrays und Kollektionen vereinheitlicht und sich auch viel eleganter schreiben lässt. Allerdings hat sie gewisse Einschränkungen, so dass man auf die alte `for` Schleife keineswegs verzichten kann. Stellen wir anhand einer Matrix die neue `for`-Schleife vor:

```
double[][] dMatrix= { {1.1,-1.5,1.8},
                      {-2.9,2.7},
                      {3.4} };

// --- der Typ der Elemente von dMatrix ist double[], der von row double
for (double[] row: dMatrix) {
  for (double element: row)
    System.out.print(element+" ");
  System.out.println();
}
```

Die Eleganz erhält die for-each dadurch, dass sie auf die übliche Laufvariable verzichtet und statt dessen direkt eine Laufvariable mit vorangestelltem Element-Typ des Containers benutzt. Bei einer Matrix ist dies zuerst eine Zeile vom Element-Typ `double[]` und für jede Zeile dann der Element-Typ `double`. Kollektionen lassen sich somit ohne explizite Angabe des Iterators elegant durchlaufen:

```
Object[] oArr= { new Date(),new Integer(2),"Hallo" };

// --- hier die wenig empfehlenswerte  nicht-generische Variante
List list= Arrays.asList(oArr);
for (Object o: list)
  System.out.println(o);
```

Eine rohe bzw. raw `Collection` wie `List` ohne Typ-Angabe seiner Elemente enthält Elemente vom Typ `Object`. Der Typ der Laufvariable ist somit `Object`. Da jeder Java-Programmierer bereits generische Kollektionen kennt, folgt noch die generische Variante einer Liste von Punkten. Dafür benötigen wir eine minimale implementierte `Point`-Klasse.

```
class Point {
  float x,y; // --- im Package Zugriff!

  public Point(float x,float y) {
    this.x= x;
    this.y= y;
  }
}
```

Die zugehörige Liste kann sehr einfach durchlaufen werden, um beispielsweise die x-Werte der Punkte zu ändern:

```
// --- eine Liste von zwei Punkten, angelegt mit Hilfe der Klasse Arrays
List<Point> pList= Arrays.asList(new Point(-1,0),new Point(3,1));

// --- ohne Cast von Object möglich:
for (Point p: pList) {
  p.x= -p.x;
  System.out.print(p.x+","+p.y+" ");   ➪ 1.0,0.0 -3.0,1.0
 }
```

Versucht man nun, alle Punkte der Liste auf den Ursprung zu setzen, erlebt man eine Überraschung.

```
Point origin= new Point(0,0);
for (Point p: pList) {
  p= origin;
  System.out.print(p.x+","+p.y+"  ");  ➪ 0.0,0.0  0.0,0.0
}

// --- Erneuter Test zeigt: keine Änderung angekommen!
for (Point p: pList) {
  System.out.print(p.x+","+p.y+" ");   ➪ 1.0,0.0 -3.0,1.0
}
```

Die Ausgabe des letzten Tests zeigt deutlich, dass mit der for-each Schleife Elemente einer Kollektion nicht neu gesetzt werden können. Präziser:

Hinweis 1.2 Besonderheiten der for-each Schleife

Mit der for-each Schleife

- können die Werte der Elemente von Arrays, deren Typ primitiv ist, nicht geändert werden.
- können die Referenzen der Elemente von Arrays und Kollektionen von Objekten nicht geändert werden.
- muss grundsätzlich das gesamte Container (Array, Kollektion, ...) durchlaufen werden.

Das Verhalten der for-each Schleife ist damit äquivalent zum Aufruf von Methoden. Auch bei Methoden werden die Argumente immer nur als Werte, d.h. als Kopie übergeben (*call by value*). Dies bedeutet, dass die Änderungen von Argumenten in der Methode nach außen unwirksam sind. Bei primitivem Typ sind dies die Werte direkt, bei Referenz-Argumente sind es die Zeiger auf die Instanzen. Somit kann man über `p` zwar den Wert von `p.x` ändern, aber nicht die Referenz `p` selbst. Die ist nach der for-each wie bei einem Methodenaufruf unverändert.

Die alte wie die neue for-Schleife lassen sich natürlich auch mit Enumerationen benutzen. Im Fall von for-each geht das besonders elegant:

```
for (Currency c: Currency.values())
  System.out.println(c);
```

Die statische Methode `values()` liefert ein Array mit allen enum-Member, das dann durchlaufen werden kann.

1.5 Variable Anzahl von Argumente

Eine variable Anzahl von Argumenten – kurz *Vararg* – ist neben Enumerationen allen C-Programmierern bekannt und wurde schon im letzten Beispiel bei der statischen Methode `Arrays.asList()` verwendet. Benötigt man eine variable Anzahl von Argumenten, schreibt man statt der Methode

```
resultType myMethod (AnyType[] args)   { /* ... */ }
```

die Varargs-Variante, wobei man die eckige Klammer einfach durch drei Punkte ersetzt:

```
resultType myMethod (AnyType... args)  { /* ... */ }
```

Ein Vararg ist nur als letzter Parameter erlaubt, da es sonst zu Zweideutigkeiten beim Aufruf der Methode kommen kann.

Hinweis 1.3 Einsatz des Vararg-Parameters

- Benötigt eine Methode als letzten Parameter ein Array, sollte man statt eines Arrays immer ein Vararg wählen.
- Innerhalb der Methode wird ein Vararg wie ein Array behandelt.
- Beim Aufruf der Methode wird ein Array passender Länge angelegt, in das die einzelnen Argumente gespeichert werden. Wird kein Argument übergeben, hat das Array die Länge Null.

Befolgt man den Hinweis, ist man bei der Verwendung der Methode weitaus flexibler und natürlich kompatibel zu der äquivalenten Methode mit einem Array-Parameter. Der Benutzer der Methode hat jetzt Wahlfreiheit. Er kann die Methode konventionell mit einem Array als letztes Argument aufrufen oder er kann statt dessen null bis beliebig viele Argumente angeben. Selbst der Aufruf ohne Argumente ist erlaubt:

```
result= myMethod();
```

Die wohl bekannteste statische Methode `main()` kann somit wie im Listing 1.1 geschrieben werden und wird vom Compiler als Ersatz für die alte `main()` akzeptiert.

Listing 1.1 Flexibilität von Varargs

```
public class TestVarargs {

  public static double average(double... dArr) {
    if (dArr==null)
      return Double.NaN;

    // --- nur zu Test-Zwecken, siehe unten!
    System.out.print("[dArr: "+dArr.length+"] ");
    double sum= 0.;

    // --- intern ist ein Vararg ein Array
    for (double d: dArr)
      sum+=d;

    return sum/dArr.length;
  }

  public static void main(String... args) {
    double[] dArr0= { };
    double[] dArr=  { 1.,2.,3. };

    System.out.println(average(1.,2.,3.));  ➪  [dArr: 3] 2.0
    System.out.println(average(1.,2.));     ➪  [dArr: 2] 1.5
    System.out.println(average(1.));        ➪  [dArr: 1] 1.0
    System.out.println(average(dArr));      ➪  [dArr: 3] 2.0
    System.out.println(average(dArr0));     ➪  [dArr: 0] NaN
    System.out.println(average(null));      ➪  NaN
    System.out.println(average());          ➪  [dArr: 0] NaN
  }
}
```

Das Listing beweist, dass die Methode `average()` sehr flexibel verwenden werden kann. Die letzte Ausgabe zeigt, dass ein Aufruf von `average()` ohne Argument nicht gleichbedeutend mit einem Aufruf `average(null)` ist, sondern mit einem Array der Länge Null.

Hätte man statt eines Varargs ein Array-Parameter genommen, könnte man `average()` auch nur mit einem `double`-Array oder `null` verwenden. Das macht verständlich, dass auch Sun in der gesamten Plattform alle letzten Parameter vom Typ Array gegen Typ Var-args ausgetauscht hat.

1.6 Autoboxing

Betrachtet man Java aus der Sicht der reinen Objektlehre – sozusagen aus der Ecke von Smalltalk & Co. – so stolpert man immer wieder über die Unterschiede zwischen primitiven Typen und Referenz-Typen. Nicht nur, dass diese beiden Arten von Typen in der Semantik unterschiedlich sind, störend ist vielmehr die häufig vorzunehmende Umwandlung von/nach den Wrapper-Typen wie beispielsweise `int` ⇔ `Integer`. Besonders unangenehm ist die Arbeit mit Kollektionen.

C# – bei seiner Geburt ein Java-Dolly – begegnete dieser Schwierigkeit von Anfang an mit *Autoboxing*. Dieser Begriff steht für die automatische Umwandlung eines primitiven Typs in sein zugehöriges Wrapper-Typ. Die Umkehrung, also vom Wrapper zurück zum zugehörigen primitiven Typ nennt man folglich *Unboxing*. Hier hat nun Java 5 nachgezogen und macht die Umwandlung – sofern notwendig – durch eine Intervention des Compilers transparent. Das soll mit Hilfe der Methode `checkBoxing()` demonstriert werden.

Listing 1.2 Autoboxing und Unboxing

```
public class TestBoxing {
  public static int checkBoxing (Number n) {
    // --- Liefert nur den Namen der Klasse und setzt n != null voraus
    String s= n.getClass().getName();
    System.out.print(s.substring(s.lastIndexOf('.')+1)+": ");

    if (n instanceof Integer)
      return (Integer)n;
    if (n instanceof Double && Double.isNaN((Double)n))
      return (Integer)null;
    return n.intValue();
  }

  public static void main(String... args) {
    System.out.println(checkBoxing(1));          ➪ Integer: 1
    System.out.println(checkBoxing(1.0));        ➪ Double: 1
    System.out.println(checkBoxing(1f));         ➪ Float: 1
    System.out.println(checkBoxing((byte)128));  ➪ Byte: -128
    System.out.println(checkBoxing(Double.NaN)); ➪ Exception...
  }
}
```

Die Methode `checkBoxing()` prüft die Fähigkeit des Compilers, anhand des primitiven Typs den passenden Wrapper-Typ zu wählen und dann das Autoboxing vorzunehmen. Wählt der Compiler ein `Integer`-Typ muss anschließend das `Integer`-Objekt wieder in eine `int` unboxed werden.

Als letztes wird das Verhalten beim Unboxing eines `null`-Werts getestet. Dies wird grundsätzlich mit einer `NullPointerException` zur Laufzeit beantwortet. Die letzte Anweisung löst somit eine Ausnahme aus. Eine Alternative zur Ausnahme wäre sicherlich ein Wert wie `NaN`, da er logisch äquivalent zu `null` ist. Der steht aber leider nur für `Double` und `Float` zur Verfügung. Beim Unboxing zeigt sich mithin der entscheidende Unterschied von primitiven Typen und zugehörigen Referenztypen. Referenzen können `null` sein und dann ist ein Unboxing unmöglich. Als vorsichtiger Programmierer muss man daher ein stillschweigendes Unboxing vorsichtshalber immer in ein `try-catch` einschließen. Vergisst man das, weil die Automatik so schön einfach ist, kann das Programm, zumindest aber die Thread abrupt beendet werden.

1.7 Überblick über Mustang-Features

Wie bereits eingangs erwähnt, bietet Java 6 keine aufregenden Spracherweiterungen, sondern nur eine Unzahl von Verbesserungen in der Plattform bzw. im API. Dies war dringend notwendig, da mit den Neuerungen in Java 5 viele neue Baustellen entstanden. Neben kleinen Fehlerbereinigungen wurden alleine wegen Generics und Annotationen unzählige Anpassungen im Detail notwendig. Alle Features und Verbesserungen wurden unter dem so genannten *Umbrella JSR 270* zusammengefasst und das ist eine respektable Liste. Die folgende Aufzählung enthält nur die wesentlichen Verbesserungen und Ergänzungen zum Java SE5, ohne auf die Java Enterprise-Edition *Java EE* einzugehen.

- **GUI**
  Das AWT wurde besser an das jeweilige Betriebssystem angepasst (Splash-Screen, System-Tray, Dialoge, etc.). Threads – ein Problem für jede GUI wie SWT oder Swing – wurde mit `SwingWorker` vereinfacht. Einzelne Komponenten wie `JTable` wurde im Detail verbessert (beispielsweise bessere Sortier/Filter-Möglichkeiten).
- **XML und Web-Service**
  Mit StAX wurde ein neues Streaming API eingeführt, dass gegenüber SAX den Vorteil hat, dass es ein Pull-Parser ist, der besser aus der Anwendung gesteuert werden kann. Mit JAX-WS 2.0 wurden die neuesten Web-Service Standards wie SOAP 1.2 und WSDL 2.0 implementiert. Interessant ist die Integration in die Java Standard Plattform 5 und nicht wie bisher nur in Java EE.
- **Datenbank**
  Mit JDBC 4.0 wird ein einfacheres API für den Datenbankzugriff ausgeliefert, was auch eine Unterstützung für XML enthält. Die große Überraschung ist allerdings, dass man mit Apache Derby eine in Java geschriebene RDBMS mit Java SE6 ausliefert.
- **Monitoring und Management**
  Speziell zur Überwachung laufender Applikationen gibt es ein verbessertes Monitoring und Management in Java 6.
- **Annotationen**
  Die Unterstützung für Annotationen wurde durch ein Pluggable Annotation-Processing

API verbessert. Somit können Annotationen Dritter zur Laufzeit besser ausgeführt werden. Annotationen bekommen damit immer mehr den Charakter von Spracherweiterungen, was vielleicht nicht unbedingt im Sinne der Erfinder ist.

- **Scripting Sprachunterstützung**
  Die Interaktion zwischen Script-Sprachen und Java war Anlass für ein JSR 223 „Scripting for the Java Platform". Als wesentliche Sprachen zählen hierzu Python, Ruby und vor allem JavaScript zur Interaktion mit Web-Browsern. Den Hype, den AJAX bzw. Google mit desktop-ähnlichen Applikationen im Browsern ausgelöst haben, führte wahrscheinlich zu diesem speziellen API.

Nahezu alle oben genannten Punkte erweitern den Core und füllen einzeln ganze Kapitel. Deshalb beschränken wir uns hier bewusst auf die einfachen API-Anpassungen, die mit Java 6 zum Standardrepertoir jedes Java-Programmierer gehören sollten. Dazu zählen Erweiterungen bei Floating-Points, bei Arrays und Containern im Package `java.util` bzw. `java.util.concurrent` und eventuell den „Pseudo"-Verbesserungen in `java.io`. Da XML, Monitoring, Scripting und vor allem Annotationen aber definitiv zu den wichtigen Techniken gehören, werden sie im zweiten Teil des Buches angesprochen.

1.8 Neue I/O-Methoden

Bevor die minimalen Anpassung im Package java.io angesprochen werden, ein kurzer Abriss zur Vorgeschichte! Das gesamte I/O-System ist seit Java 1.0 eine große Baustelle. Zuerst gab es nur eine Input-/Output-Hierarchie für Streams, bis man recht schnell feststellte, dass diese Hierarchie Unicode-Zeichen nicht wirklich gut unterstützt. Also wurden `Reader`- und `Writer`-Hierarchien für Unicode-Streams hinzugefügt. Die Hierarchien der Zeichen-Streams existieren seitdem parallel zu den byte-orientierten Streams. Sie haben natürlich viele Gemeinsamkeiten wie low-level und high-level Streams und funktionieren beide nach dem *Decorator-Pattern*. Man baut sich folglich die gewünschte Stream-Instanz durch Ineinander-Schachteln von Konstruktoren geeigneter I/O-Klassen zusammen. Im Stil der 90iger Jahre war das Framework nahezu klassen-basiert aufgebaut, um zukünftige Änderung möglichst anspruchsvoll, besser gesagt unmöglich werden zu lassen.

Die offensichtlichen Defizite wurden schließlich so groß, dass man sich in Java 1.4 entschloss, ein interfaces-basiertes *NIO* in die Standard-Edition aufzunehmen. Aber nicht anstatt des alten APIs, sondern zusätzlich! Seitdem arbeitet der ambitionierte Java-Entwickler konzeptionell mit Stream-Klassen und Channel-Interfaces. Dieses System inklusive Encoding/Decoding-Änderungen, Memory-Mapping und Threading ist recht komplex. In vielen Unternehmen wird deshalb das NIO entsprechend respektvoll beäugt. Alle warten nun auf den „großen Wurf". Man ahnt es, Java 6 ist es leider nicht! Die Sache ist wieder einmal auf die kommende Version 7 alias Dolphin verschoben. Aber zugegeben, wie im Automobilbau können Detailverbesserungen an einem alten Modell trotzdem funktional sein.

1.8.1 File

Ergänzungen findet man in der uralten Java 1.0 Klasse `File`. `File` hat nichts mit der oben erwähnten Stream-Hierarchie zu tun. Es ist eine singuläre Klasse, deren Sinn darin besteht, Datei- und Verzeichnisnamen des jeweiligen Betriebssystem zu abstrahieren. Anstatt mit Strings soll mit `File` gearbeitet werden, womit `File` aber alle möglichen Datei- und Verzeichnisdienste anbieten muss. Und genau da fehlten einige, die nun hinzugefügt wurden.

Listing 1.3 Speicherplatz berechnen in Java 6

```
public class TestFile {

  public static void main(String... args) {
    // --- holt die Partition:
    File f= new File("/");

    // --- gesamter verfügbarer Speicherplatz
    System.out.println(f.getTotalSpace());          ➪ 79682387968
    // --- der noch frei Speicherplatz (ohne Garantie):
    System.out.println(f.getFreeSpace());           ➪ 54356234240
    // --- der freie Speicherplatz, der von der JVM benutzt werden kann
    System.out.println(f.getUsableSpace());         ➪ 54094090240

    // --- Ein USB-Stick
    f= new File("/volumes/hta_stick");
    // --- Liefert benutzen Speicherplatz oder 0, wenn Stick nicht eingesteckt
    System.out.println(
      f.getTotalSpace()-f.getFreeSpace());          ➪ 106496

    f= new File("/System/Library/Frameworks/"+
          "JavaVM.framework/Versions/1.6.0/commands/javac");
    // --- Ist javac ausführbar?
    System.out.println(f.canExecute());             ➪ true
  }
}
```

Die ersten drei Methoden berechnen Speicherplatz in einer Partition. Ein physikalischer Speicher wird bekanntlich in ein oder mehrere Partitionen unterteilt. Jedes Verzeichnis bzw. jede Datei gehört zu einer Partition. Eine sicherlich nützliche Information ist die Größe und der freie Speicherplatz einer Partition. Es gilt die folgende Beziehung:

```
getTotalSpace() >=  getFreeSpace() >=  getUsableSpace()
```

Unter MAC OS X wird unterhalb von `/volumes` ein USB-Stick eingebunden. Beim Vergleich mit der OS X-Information zum `hta_stick`-Stick stellt man Übereinstimmung fest (siehe Abbildung 1.2).

Abbildung 1.2 Berechnung des belegten Speicherplatzes

Die letzte Anweisung in Listing 1.3 testet auf Ausführbarkeit des Java 6 Compilers unter Mac OS X. Zur Vollständigkeit noch die restlichen neuen Methoden in Mustang, wobei das Subskript $_{opt}$ für optional steht. Da es in Java keine optionalen Parameter gibt, wird dies durch Überladen der Methoden erreicht. Im folgenden Fall also eine Methode nur mit dem ersten Parameter und eine weitere mit beiden:

```
boolean setExecutable(boolean executable,
                      boolean_opt ownerOnly);

boolean setReadable(boolean readable,boolean_opt ownerOnly);

boolean setWritable(boolean writable,boolean_opt ownerOnly);
```

Der Parameter `ownerOnly` bestimmt, ob sich das Setzen nur auf den Besitzer (`true`) oder auf alle Benutzer (`false`) beziehen soll. Will man die `set`-Operation nur auf den Besitzer beziehen, sind die Setter ohne `ownerOnly` besser, da dann automatisch `ownerOnly` auf `true` gesetzt ist. Nur im Fall, dass der Setter erfolgreich ausgeführt wird, wird als Ergebnis `true` zurückgeliefert. Gibt es im Betriebssystem keinen Unterschied zwischen Besitzer und allen Benutzern, so wirkt der Setter grundsätzlich auf alle. Ist ein `SecurityManager` installiert, kann im Fall einer Sicherheitsverletzung auch eine `SecurityException` ausgelöst werden.

1.8.2 Console-I/O

Ein passender Untertitel zu diesem Kapitel wäre an sich: *Das Grauen hat einen Namen*! Denn etwa genauso lange wie es `System.in` gibt, wurde diese Art der Eingabe über die Konsole von Entwicklern heftigst kritisiert. In 1997 wurde eine offizielle Anfrage auf Verbesserung *RFE* (*R*equest *F*or *E*nhancement) eingereicht. Sun reagiert in Mustang mit Methoden einer neuen Klasse `java.io.Console`, die den Forderungen nun nachkommt.

Wirklich? Angesichts der vergangenen acht Jahren und der Art, wie die neuen Methoden implementiert sind, kommen Zweifel auf, ob Sun überhaupt eine Verbesserung wollte. Denn von C her gibt es schon lange das, was gefordert wurde und an sich sollte eine Übertragung auf Java im Rahmen des Möglichen liegen. C bietet Methoden wie `getch()`, `getche()` oder `kbhit()` bzw. einer curses-Bibliothek unter Unix. Kann also prinzipiell nicht so schwer sein, denkt man sich.[2] Zeigen wir nur die beiden Kern-Methoden von `Console` zur Abschreckung. Es ist das, was die Sun-Ingenieure nach zehn Jahren implementiert haben.

Listing 1.4 Die „ultimative" Methode console.readline()

```
public class TestConsole {
  public static void main(String... args) {
    Console console = System.console();
    if (console==null) {
      System.out.println ("Console nicht verfügbar");
      System.exit(1);
    }

    // --- die beiden Parameter von readline() haben die gleiche
    //     Funktion wie String.format() und basieren auf java.util.Formatter
    //     eine einfache Variante wäre: String input = console.readLine();
    String input=
      console.readLine("%s","Eingabe mit RETURN beenden: ");
    System.out.println(input);

    // --- eine Alternative zu readLine(), die readPassword()
    char[] pw;
    pw= console.readPassword("%s", "Gib Passwort:");
    System.out.println(pw);
  }
}
```

Zuerst muss man sich von `System` die Konsole holen. Arbeitet man von einem Terminal-Fenster – vorausgesetzt, dass keine Umleitung der Standard-/Ausgabe gemacht wurde – bekommt man das `Console`-Objekt. Ansonsten – innerhalb einer IDE wie Netbeans – erhält

[2] Zumal sich der alte Programmierer gerne auch an Turbo Pascal der 80iger Jahre erinnert, wo Console-Input der Standard war.

man `null`. Nun kann man sich ähnlich zu dem Stream `BufferedReader` mit einer der beiden `readLine()`-Methoden eine Eingabezeile als `String` holen. Das erfolgt immer mit Eingabe-Echo, ohne geht es nicht! Es werden also alle Zeichen auf der Tastatur – zumindest bei Mac OS X – inklusive der Umlaute und Sonderzeichen wie eckige Klammern akzeptiert.

Verschreibt man sich bei einem der nicht-ASCII-Zeichen, hört der Spaß allerdings auf. Denn korrigiert man seinen Fehler und löscht nach hinten – standardmäßig von allen Konsolen unterstützt – löscht man zwar in der Konsole das Zeichen, beispielsweise ein `'ä'`, aber im Zeichenpuffer von Java bleibt ein Rest stehen, der bei dem Ausdruck der Eingabe mit der Ausgabe `?` angezeigt wird. Drückt man zweimal die Löschtaste, ist tatsächlich das `'ä'` im Ausdruck weg. Dafür fehlt in der Konsole aber das Zeichen vor dem `'ä'`! Verstanden? Nicht unbedingt! Deshalb zur visuellen Unterstützung die Abbildung 1.3. Im Terminal-Fenster wird die `main()` aus Listing 1.4 ausgeführt.

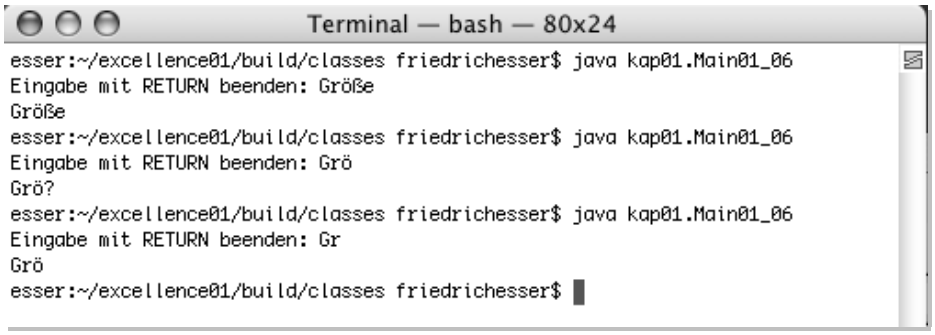

Abbildung 1.3 Eingabe mit Hilfe der Klasse Console

Zuerst wird ohne Korrektur `"Größe"` eingegeben. Die Ausgabe ist korrekt. Bei den nachfolgenden beiden Programmausführungen wurde `"Größe"` eingegeben und anschließend zweimal bzw. dreimal nach hinten gelöscht. Folglich steht dann als Eingabe bei den Programmausführungen einmal `"Grö"` bzw. `"Gr"`. Die zugehörige Ausgabe von `input` ist aber `"Grö?"` bzw. `"Grö"` und dokumentiert das oben angesprochene Problem.[3]

Bei `readPassword()` ist das Echo ausgeschaltet und der Cursor bleibt bei der Eingabe auf der Stelle hinter dem `':'` stehen. Auch nicht schlecht! Wie wäre es – zumindest optional – mit einem Stern pro Tastendruck? Da aber Passwörter vom Benutzer immer korrekt eingegeben werden und dabei Nicht-ASCII-Zeichen ohnehin verboten sind, hat man ja als Programmierer keine Probleme. Fazit: Das neue Console-IO gehört nicht zu den Sternstunden der Programmierung!

[3] Hoffen wir auf einen kompetenten Sun-Ingenieur im Jahre 2014. Die Frage ist nur, ob es dann noch steinalte Programmierer gibt, die auf ihre geliebte Konsole beharren.

1.9 Floating-Point Essentials

Floating-Point gibt es wie I/O auch bereits seit der Version 1.0. Sie folgen dem IEEE 754 Standard, der aber immer nur in Teilen umgesetzt wurde. In Java 6 sind wieder einige Empfehlungen der IEEE 754 umgesetzt worden. Da Gleitkommazahlen insbesondere in der Ingenieurwissenschaft intensiv genutzt werden, ist eine Auffrischung des Standardwissens auf den Stand Java 6 sehr wichtig.

Es gibt leider nur zwei primitive Floating-Point Typen `float` (32 Bit) und `double` (64 Bit), die sich am Standard IEEE 754 orientieren. Passende Methoden zu beiden Typen findet man dann in der Klasse `Math`. Für die Darstellung der Neuerungen reicht in der Regel ein Typ, da sie für den anderen analog ist. In vielen Fällen wählen wir den Typ `double`, da sich die Werte ohne Suffix `f` einfacher schreiben lassen. Weiterhin wird bewusst auf „die große Theorie“ zugunsten von Beispielen verzichtet.[4]

1.9.1 Floating-Points vs. Reelle Zahlen

Lernt man als Ingenieur eine Programmiersprache wie Java, besteht eine erste Erfahrung darin, den Unterschied zwischen mathematischen Zahlen mit zugehörigen Operationen und der Übertragung in Java-Code zu erlernen. Sofern nicht, sind die vom Programm ermittelten Ergebnisse – obwohl durchaus mathematisch korrekt codiert – reine Glücksache.

IEEE 754 komprimiert

Die erste Erfahrung ist die Auseinandersetzung mit Wertebereich und Präzision von `float` und `double`. Sie sind aufgrund der Bit-Länge unterschiedlich, wobei aber ihr interner Aufbau sehr ähnlich ist:

`float`
`1 Bit Vorzeichen, 8 Bit Exponent und 23 Bit Mantisse`
`Wertebereich: {0,±1.4·10⁻⁴⁵ … ±3.40·10⁺³⁸} ∪ {±∞} ∪ {NaN}`
`double`
`1 Bit Vorzeichen, 11 Bit Exponent und 52 Bit Mantisse`
`Wertebereich: {0,±4.9·10⁻³²⁴ … ±1.80·10⁺³⁰⁸} ∪ {±∞} ∪ {NaN}`

Alle Werte werden normiert abgelegt. Die Zahl Null wird beispielsweise als spezielles Bitmuster `0...0` des Exponenten gespeichert und existiert je nach Vorzeichen als `-0` und `+0`. Der Exponent `1...1` signalisiert einen der drei Werte `±Unendlich` oder `NaN` (*Not-a-Number*). `NaN` steht je nach Interpretation für eine ungültige oder für keine Zahl.

Aufgrund der Größe der Mantisse lassen sich maximal 9 bzw. 17 Ziffern/Stellen einer Dezimalzahl in einer `float` bzw. `double` ablegen. Das nennt man *Präzision* (*accuracy*). Sie ist nicht zu verwechseln mit der Genauigkeit einer Berechnung, die völlig „aus dem Ruder“ laufen kann. Besteht eine Zahl aus mehr Stellen als die Präzision erlaubt, werden Stellen au-

[4] Siehe hierzu weiterführende Literaturangaben/Links am Ende des Kapitels.

ßerhalb des Bereichs einfach ignoriert. Aber selbst, wenn die Anzahl der Stellen innerhalb der Präzision liegen, ist Vorsicht geboten.

Repräsentation

Von der Stellenanzahl her ist ein Wert wie `0.1` wohl kein Problem, wenn da nicht die Konvertierung zwischen Dezimal und Binär wäre. Denn die führt zu vergleichbaren Problemen, wie die bei der Umwandlung von rationalen Zahlen `a/b` in Dezimalzahlen. Beispielsweise lässt sich die rationale Zahl 1/3 nicht exakt als endliche Dezimalzahl darstellen. Und genau das gilt auch für viele Dezimalzahlen wie `0.1`, die keine exakte binäre Repräsentation haben. Bei Überschreitung des normalen Wertebereichs bzw. bei ungültigen Operationen spielen die besonderen Werte `Infinity` und `NaN` ein wichtige Rolle. Vor allem `NaN` zerstört das normale Verhalten von reellen Zahlen, die mathematisch gesehen total geordnet sind.

Damit sind die wesentlichen Unterschiede zwischen Floating-Point und reellen Zahlen zwar angesprochen. Aber es es fehlen noch Details und Beispiele.

Präzision am Beispiel

Präzision kann schon an einem einfachen Beispiel demonstriert werden:

```
System.out.println(1E16 + 1);    ➪ 1.0E16
```

Die Zahl `10000000000000001` überschreitet mit 17 Stellen die Präzision eines `double`, womit die Addition von `1` wirkungslos ist. Verschiedene dezimale Werte – als Dezimal-String geschrieben – führen zu gleichen Floating-Points, wenn sie erst außerhalb der Präzison gleich sind:

```
float f= 1.00000001f;

System.out.println(f);                          ➪ 1.0
System.out.println(
    Float.parseFloat(Float.toString(f))==f);    ➪ true
```

Wie man sieht, trifft der dezimale (String-)Wert von `f` in der ersten Ausgabe nicht den zuerst angegebenen Wert von `f`. Aber eine reine Konvertierung binär → dezimal → binär liefert immer dieselbe Zahl.

Dezimal-Binär-Konvertierung

Die Konvertierung von dezimalen Werten in interne binäre Werte führt zu Problemen, wie das folgende Code-Fragment zeigt.

```
double d= 0.3;
while (d != 0.0) {
  if (d<0) {
```

```
      System.out.println(d);            ➪ -2.7755575615628914E-17
      break;
    }
    d-= 0.1;
  }
  System.out.println(0.1+0.1+0.1);      ➪ 0.30000000000000004
```

Die Ausgabe bestätigt, dass `0.1` in binärer Form nicht exakt abgespeichert werden kann. Bereits einfache Additionen und Subtraktionen können zu mathematisch inkorrekten Werten führen. Ohne `break` hätte man oben eine Endlosschleife programmiert.

Hinweis 1.4 Test auf Gleichheit, Ungleichheit

- Vergleiche mit `==` und `!=` können aufgrund von Ungenauigkeiten bei Floating-Point zu falschen logischen Werten führen. Sie sind deshalb grundsätzlich zu vermeiden und durch einen Intervall-Test zu ersetzen (siehe Abschnitt 1.9.2).

Aufgrund der nicht exakten Umwandlung gibt es auch Überraschungen bei der Umwandlung von `float` in `double`:

```
float f= 0.2F;
System.out.println(Float.toString(f));       ➪ 0.2
System.out.println(Double.toString(f));      ➪ 0.20000000298023224

double d= 0.2;
System.out.println(d==f);                    ➪ false
```

Hier führt der Wert `0.2` auf unterschiedliche `float` und `double` Mantissen-Werte. Zwischen zwei benachbarten `float`-Werten liegen nun einmal über ein halbe Milliarde `double`-Werte, genauer:

$2^{52-23} = 2^{29} = 536$ Millionen

Diese `double`-Werte werden alle durch den unteren/oberen `float`-Wert repräsentiert.

Spezielle Werte Infinity, NaN

Neben dem normalen Wertebereich existieren im Gegensatz zu integralen Typen wie `int` und `long` noch spezielle Werte:

```
System.out.println(Double.MAX_VALUE * 1.1);    ➪ Infinity
System.out.println(1.0/-0.);                   ➪ -Infinity

System.out.println(0./0);                      ➪ NaN
System.out.println(5.%0);                      ➪ NaN
```

```
System.out.println(Float.POSITIVE_INFINITY+
                   Float.NEGATIVE_INFINITY);    ➪ NaN
System.out.println(Float.POSITIVE_INFINITY*0); ➪ NaN
```

Die `Infinity`-Werte resultieren immer aus einem Überlauf des Wertebereichs. Ungültige Operationen führen dagegen zu `NaN`. Diese drei speziellen Werte können nie wieder durch eine Operation in den normalen Wertebereich überführt werden. Ein Versuch führt automatisch zu `NaN`, wie die letzten beiden Ausgaben zeigen. Insbesondere kann `NaN` nie wieder verlassen werden.

Hinweis 1.5 NaN vs. totaler Ordnung

- `NaN` zerstört die totale Ordnung von Floating-Points. Mit Ausnahme von `NaN` gilt für zwei Zahlen `d1`, `d2` immer, dass genau einer der Vergleiche `true` ist:

  ```
  d1>d2 oder d1==d2 oder d1<d2
  ```

Für `NaN` sind alle drei Vergleiche `false`. Damit wird der schönste Sortier-Algorithmus für `float` oder `double` zerstört, sofern er nur auf dem Vergleich mit `<`, `>` oder `==` basiert. Das hat Sun natürlich bei ihren Sortier-Algorithmen in `Arrays` berücksichtigt:

```
double[] dArr={2.0,Double.NaN, -1.,Double.NaN,-100.,
               100.,Double.POSITIVE_INFINITY,Double.NaN};
Arrays.sort(dArr);

System.out.println(Arrays.toString(dArr));
          ➪ [-100.0, -1.0, 2.0, 100.0, Infinity, NaN, NaN, NaN]
```

Die `NaN`-Werte werden von Sun als größte Werte oberhalb von Unendlich eingeordnet. Wie diese Sortierung vorgenommen wird, ist sicherlich auch interessant.[5]

Verlust von Genauigkeit

Um ein Gefühl für Genauigkeit zu bekommen, betrachtet man am besten eine mathematisch einfache Funktion wie :

$$calc(n) = n!e^n/n^n$$

Die einzelnen Faktoren `n!`, e^n und n^n überschreiten selbst für kleine `n` sehr schnell den normalen Wertebereich. Klüger ist also folgende Art von Berechnung, hier am Beispiel von `n=5` demonstriert:

$$5!e^5/5^5 = e/5 * 2e/5 * 3e/5 * 4e/5 * 5e/5$$

Die einzelnen Faktoren sind immer im „grünen Bereich“ und für relativ große `n` kommt man noch zu brauchbaren Ergebnissen. Es gibt zwei einfache Möglichkeiten, diesen Ausdruck

[5] Und kann leicht in den Sourcen zu `Arrays` eingesehen werden.

mit Hilfe einer `for`-Schleife zu berechnen, aufsteigend oder absteigend. Da die Anforderung an die Präzision nicht hoch ist, wählen wir den Typ `float` für die Berechnung (Anmerkung: statischer Import der Klasse `Math`):

```
// --- die Variation mit Typ double führt zu der äquivalenten Methode calcD()
static float calc1F(int n) {
  float res= 1F;
  // --- Math.E ist zwar eine double, wird aber aufgrund von *= nach float konvertiert
  for (int i= 1; i<=n; i++)
    res*= i*E/n;
  return res;
}

// --- die absteigende Variante:
static float calc2F(int n) {
  float res= 1F;
  for (int i= n; i>=1; i--)
    res*=i*E/n;
  return res;
}
```

Ein erster Test mit `n= 100` liefert wie erwartet Übereinstimmung:

```
System.out.printf("%4.2f\n",calc1F(100)); ➪ 25,09
System.out.printf("%4.2f\n",calc2F(100)); ➪ 25,09
```

Als nächstes soll das Ergebnis einmal für `n= 295` berechnet werden. Hier kommt es zu einer krassen Abweichungen der Ergebnisse:

```
System.out.printf("%4.2f\n",calc1F(295)); ➪ 33,42
System.out.printf("%4.2f\n",calc2F(295)); ➪ Infinity
```

Der erste Wert scheint noch ok, der zweite ein Overflow oder Fehler zu sein. Also wechselt man zur Überprüfung den Typ `float` den `calc1F()` gegen Typ `double` aus. Die `double`-Variante `calcD()` führt zum Ergebnis :

```
System.out.printf("%4.2f\n",calcD(295));  ➪ 43,06
```

Das erste Ergebnis ist somit auch falsch!

Fazit
Bei der Verwendung des Typs float wird man sehr schnell ein Opfer der Genauigkeit. Besser, man beherzigt die folgenden Regel.

Hinweis 1.6 Floating-Point Regeln

- Verlust von Präzision aufgrund von Dezimal-Binär-Konvertierung und (einfachen dezimalen) Operationen.
- Berechnungen sollten grundsätzlich mit dem Typ `double` durchgeführt werden.
- Der Typ `float` ist nur zum Speichern von Dezimalzahlen nützlich, deren Werte keine hohe Genauigkeit benötigen.
- Berechnungen sollten so programmiert werden, dass sie einen Over- bzw. Underflow signalisieren und nicht einfach nur ein falsches Ergebnis liefern.

Bei Geldwert-Berechnungen wird man prinzipiell bereits am ersten Punkt scheitern. Die Dezimal-Binär-Konvertierung ist zu ungenau. Die Wahrscheinlichkeit ist hoch, Geld zu erschaffen oder zu vernichten. Der zweite und dritte Punkt ist einfach umzusetzen. Die Methode wie `calcF()` ist durch `calcD()` zu ersetzen.

Problem Genauigkeit
Ein wirklich hartes Problem bereitet die Frage nach der Genauigkeit, angesprochen im ersten und vierten Punkt. Alle vier oben verwendete Arten der Programmierung `calc1F()` ... `calc2D()` sind jedenfalls reichlich naiv. Es gibt nichts, was signalisiert, dass das Ergebnis nicht mehr korrekt ist. Da hilft auch eine Umstellung auf `double` wenig. Denn sie verschiebt nur die Fehlergrenze!

Es gibt leider keine allgemeine Methode sicherzustellen, ob und wie genau das Ergebnis ist. Das ist von der Berechnung abhängig. Es gibt aber zumindest eine Art der konservativen Programmierung, die in Abschnitt 1.9.3 nach den nun folgenden Neuerungen in Java 6 vorgestellt werden soll.

1.9.2 Neue Konstante und Methoden in Java 6

Bei Java 6 wurden neue Konstante für `float` bzw. `double` eingeführt:

Neue Java 6 Konstante	*float*	*double*
SIZE	32	64
MIN_EXPONENT	-126	-1022
MAX_EXPONENT	127	1023
MIN_NORMAL	1.17549435E-38F	2.2250738585072014E-308

`MIN_NORMAL` ist die interessanteste der Konstanten, da bis zu diesem Wert die „normale Mathematik“ gilt. Zwischen `MIN_NORMAL` und `MIN_VALUE` regiert der Zufall.

Erhärten wir dies mit einem Beispiel:

```
System.out.println(7.5e-324);           ➪ 1.0E-323
System.out.println(7.4e-323<7.5e-323); ➪ false
System.out.println(7.4e-308<7.5e-308); ➪ true
```

Der letzte Vergleich liegt oberhalb von `MIN_MORMAL`, ist also korrekt. Die neuen Methoden `Math.getExponent()` liefern auch Einblick in die Speicherung der besonderen Werte Infinity, `NaN` und `0`:

```
System.out.println(getExponent(Float.POSITIVE_INFINITY)); ➪ 128
System.out.println(getExponent(Float.NEGATIVE_INFINITY)); ➪ 128
System.out.println(getExponent(Float.NaN));               ➪ 128
System.out.println(getExponent(Float.MIN_VALUE));         ➪ -127
System.out.println(getExponent(0F));                      ➪ -127
```

Für die Extremwerte des Wertebereichs werden die größten und kleinsten Byte-Werte des Exponenten reserviert.[6] Die Exponenten enthalten Werte zur Basis 2. Möchte man dagegen die Exponenten zur Basis 10 bestimmen, beispielsweise für `float.MAX_VALUE`, so muss man umrechnen:

```
System.out.println(
         (int)(log10(2)*getExponent(Float.MAX_VALUE)));  ➪ 38
```

Methoden ulp(), nextUp(), nextAfter()

Schon Java 5 führte den Begriff *Unit in the last Place* – kurz *ulp* genannt – ein. Ein ulp steht für den relativen Fehler, der mit einem Zahlenwert verbunden ist. Hat man einen beliebigen festen Wert `d`, so sind `d ± ulp` die benachbarten Zahlen von d, die dezimal unterschiedlich angezeigt werden. Werte im ulp-Intervall werden dann entweder als `d`, `d-ulp` oder `d+ulp` angezeigt.

Für integrale Typen wie `int` oder `long` hat `ulp` für den gesamten Wertebereich immer den Wert 1. Da der Wert eines ulps für Dezimalzahlen aber von der Größenordnung abhängig ist, findet man hierzu eine Methode `Math.ulp(double d)`. Bei Floating-Points steigt mit jeder 10er-Potenz der Abstand zweier benachbarter Werte um etwa das 10fache. Denn die Mantisse ist immer konstant. Ulps helfen also zur Bestimmung der *relativen Präzision*. Eng im Zusammenhang mit ulps stehen die neuen Java 6 Methoden `nextUp()` und `nextAfter()`. Für `d >= 1` gilt die mathematische Beziehung:

```
ulp(d) = nextUp(d) - d
```

Bei der Methode `nextAfter(double d, double direction)` kann man noch mit der Richtung `direction` bestimmen, welche der beiden benachbarten Werte man will. Interessant ist aber an sich nur `nextUp()`, da die Methode auch wesentlich schneller ist.

---

[6] Wobei nach IEEE 754 der Wertebereich `-127...128` ist, was von dem des Java-Typs `byte` abweicht.

```
System.out.println(ulp(1.0)+" , "+(nextUp(1.0)-1.0));
                    ➪ 2.220446049250313E-16 , 2.220446049250313E-16
System.out.println(ulp(123E8)+" , "+(nextUp(123E8)-123E8));
                    ➪ 1.9073486328125E-6 , 1.9073486328125E-6
```

Warnung: Unterhalb von `MIN_NORMAL` hat die normale Mathematik keine Gültigkeit. Beispielsweise ist das *Assoziativ-Gesetz* `(a*b)*c= a*(b*c)` verletzt:

```
System.out.println(Double.MIN_VALUE == ulp(0.));     ➪ true
System.out.println(ulp(0.)*(1./ulp(0.)));            ➪ Infinity
```

Da `ulp(0.)` gleich `MIN_VALUE` ist, sollte man damit nicht rechnen. Erst `MIN_NORMAL` liefert das normale mathematische Verhalten:

```
System.out.println(
         Double.MIN_NORMAL*(1./Double.MIN_NORMAL)); ➪ 1.
```

Hinweis 1.7 Basis-Arithmetik und Math-Funktionen

- Bei allen Berechnungen oder Funktionen der `Math`-Klasse ist ein Ergebnis mit einem Fehler `<= ulp/2` optimal. Denn im ungünstigsten Fall liegt ein Ergebnis `ulp/2` vom nächsten darstellbaren Wert `d` entfernt, d.h.:
  `ulp(resultat) <= ulp(d)/2`
- Für die Basis-Arithmetik sowie die meisten Funktionen in `Math` ist der Fehler eines Ergebnisses `<= ulp`.

Werden zwei dezimale Zahlen `d1` und `d2` auf gleich bzw. ungleich getestet, ist nach dem zweiten Punkt des Hinweises zu prüfen, ob einer der beiden Werte in einem ulp-Intervall des anderen liegt. Leider testet `Double.equals()` die Gleichheit wie folgt:[7]

```
d1.doubleValue()== d2.doubleValue()
```

Deshalb benötigt man zusätzlich ein toleranteres `equals()`:

```
// --- NaN-Test fehlt!
static boolean equals(double d1,double d2) {
  return abs(d1 - d2) <= (ulp(d1)>ulp(d2)? ulp(d1): ulp(d2));
}
```

Die Methode liefert `true`, wenn aus der Sicht beider Werte der Abstand `<=ulp` ist. Im folgenden Beispiel wenden wir dieses `equals()` auf Basis-Operationen mit dem binär sehr ungünstigen Wert `0.1` an (siehe hierzu auch den Abschnitt Dezimal-Binär-Konvertierung oben).

---

[7] Da dies durch eine Bit-Vergleich gemacht wird, bilden `NaN` und `±0` die Ausnahmen (siehe Sourcen von Double.equals()).

```
double d1= 0.1+0.1+0.1;
double d2= 0.1 *3.;
double d3= 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1;

System.out.println(ulp(0.3));          ➪ 5.55111e5123125783E-17
System.out.println(d1-0.3);            ➪ 5.551115123125783E-17
System.out.println(equals(d1,0.3));    ➪ true
System.out.println(d2-0.3);            ➪ 5.551115123125783E-17
System.out.println(equals(d2,0.3));    ➪ true

System.out.println(ulp(1.));           ➪ 2.220446049250313E-16
System.out.println(d3-1.);             ➪ -1.1102230246251565E-16
System.out.println(equals(d3,1.));     ➪ true
```

Wie man sieht, ist der `ulp`-Abstand in diesem Fall optimal, da bei `ulp/2` nur der letzte Vergleich `true` liefern würde.

Methoden scalb(), copySign()

Die Methode `scalb(double d, int scaleFactor)` berechnet `d * 2`$^{\texttt{scaleFactor}}$ und ist i.d.R. schneller und präziser, da sie mit Potenzen zur Basis 2 arbeitet. Dies setzt voraus, dass kein Overflow stattfindet.

```
System.out.println(Math.scalb(3.,4));          ➪ 48.0
System.out.println(Math.scalb(3.,1025));       ➪ Infinity
```

Mit `copySign()` kann das Vorzeichen von einem Floating-Point auf die andere übertragen werden:

```
System.out.println(Math.copySign(3.,-1.23));    ➪ -3.0
System.out.println(
  Math.copySign(Double.NEGATIVE_INFINITY,10.)); ➪ Infinity
```

1.9.3 Floating-Point Constraints

Mit dem Begriff Constraints wird der weite Bereich des Constraint-Based-Programming verbunden. Im folgenden wird nur anhand eines kleinen Beispiels aufgezeigt, welche Bedeutung gut gewählte Restriktionen auch für Berechnungen haben können.

Die im letzten Abschnitt von 1.9.1 gewählte Funktion zum `calcD()`-Beispiel war der wesentliche Teil der *Stirlingschen* Formel. Diese Formel bietet eine Abschätzung für die Fakultät `n!` :

$$\texttt{calcS(n)} = n!e^n / n^n \sqrt{2\pi n} \rightarrow 1 \quad \text{für } n \rightarrow \infty$$

Damit ist diese Methode als Testkandidat für Constraints ideal geeignet, da man das Ergebnis für große n mathematisch vorhersagen kann. Zuerst ist es wichtig, für den Ablauf einer Berechnung überhaupt *Constraints* (Restriktionen) zu finden.

Invariante: In Floating-Point-Fall wählen wir eine sehr plausible Invariante

- Das Ergebnis jedes Rechenschritts muss im Intervall `MIN_NORMAL...MAX_VALUE` liegen (Gründe siehe letzten Abschnitt 1.9.2).

Gibt es bei Berechnungen beispielsweise aufgrund des Assoziativ-Gesetzes Alternativen, muss man die wählen, deren Ergebnis die Invariante einhält. Bei der Stirlingschen Formel gibt es grundsätzlich nur zwei Alternativen. Die Multiplikation von unten mit aufsteigenden i-Werten führt zuerst zu sehr kleinen (Zwischen-)Ergebnissen, die mit absteigenden i-Werten führt dagegen zuerst zu sehr großen. Dies kann man ausnutzen:

```
static double calcS(int n) {
  double res= 1.0, rold;
  double fac= E/n;
  for (int i=1,j=n; i<=j;) {
    // --- rold hält den Wert für 2. Alternative fest
    rold= res;

    // --- 1. Alternative: Prüfen einer Multiplikation „von unten“
    if ((res*=i*fac) > Double.MIN_NORMAL)
      i++;
    // --- 2. Alternative: Prüfen einer Multiplikation „von oben“
    else if ((res=rold*j*fac) < Double.MAX_VALUE)
      j--;

    // --- Invarianz-Verletzung: Konsequenz Exception
    else
      // return NaN     <-- Alternative
      throw new ArithmeticException(
        "Overflow: "+res+" Index "+ i +","+ j);
  }
  return res/Math.sqrt(2*PI*n);   // liefert für n<=0  NaN
}
```

Vergleicht man dies mit der naiven Form der Berechnung:

```
static double calcD(int n) {
  double res= 1.;
  for (int i= 1; i<=n; i++)
    res*=i*E/n;
  return res/Math.sqrt(2*PI*n);
}
```

so wird man bei einem Vergleich von `calcS()` positiv überrascht:

```
System.out.printf("%4.10f\n",calcS(100));        ➪ 1,0008336779
System.out.printf("%4.10f\n",calcD(100));        ➪ 1,0008336779

System.out.printf("%4.10f\n",calcS(1000));       ➪ 1,0000833368
System.out.printf("%4.10f\n",calcD(1000));       ➪ 1,0000833368

System.out.printf("%4.10f\n",calcS(2100));       ➪ 1,0000396833
System.out.printf("%4.10f\n",calcD(2100));       ➪ 12,1640829183

System.out.printf("%4.10f\n",calcS(10000000));   ➪ 1,0000000086
```

Wie man sieht, läuft bei `calcD()` der Wert etwa bei `n= 2100` „aus dem Ruder“. Die Methode `calcS()` arbeitet dagegen noch bei zehn Millionen zuverlässig und bestätigt damit auch die Stirlingsche Formel.

Hinweis 1.8 Früh-Diagnose von Fehlern mit Hilfe von Invarianten

- Liegt ein Resultat einer Berechnung im erlaubten Wertebereich, kann es keinen Fehler mehr signalisieren und wird akzeptiert. Deshalb müssen Fehler mit Hilfe von *Pre-Conditions* (erlaubter Wertebereich von Variablen) und Invarianten, die jede Operation einzuhalten hat, frühzeitig signalisiert werden.

Je nach Bedarf kann ein Fehler bei einem Floating-Point-Ergebnis als `NaN` oder als Ausnahme signalisiert werden.

1.10 Typsichere Array-Kopien

Im Gegensatz zu Listen sind Arrays von fixer Größe. Sie sind extrem schnell und angenehm zu verwenden, da sie in die Sprache eingebaut sind.[8] Sehr unangenehm sind dagegen Anpassungen des Arrays in der Größe, sei es, weil das Array zu groß oder zu klein gewählt wurde. Die Anpassung mit Hilfe von `System.arraycopy()` ist nicht unbedingt intuitiv und durchaus fehleranfällig. Deshalb gibt es nun in Java 6 sehr effiziente Kopiermethoden. Diese Methoden sind für jeden primitiven Typen überladen:

primitiveType`[] copyOf(`*primitiveType*`[] originalArray, int newLength);`

Der Typ *primitiveType* steht wie üblich für `byte`, ... , `double`. Zusätzlich gibt es noch eine einfache generische Versionen, die an Stelle von *primitiveType* eine Typevariable `T` verwendet:

```
T[] copyOf(T[] originalArray, int newLength);
```

[8] Deshalb wird ihre Schreibweise wahrscheinlich in Java 7 auch auf Kollektionen übertragen werden.

Gegenüber der Alternative `Object[]` ist die Variante `T[]` analog zu `List<T>` typsicher. Für `T` kann jeder beliebige Typ eingesetzt werden. Eine weitere Version ist ebenfalls generisch, sieht jedoch ohne Generics-Kenntnisse recht abschreckend aus:[9]

```
public static <T,U> T[] copyOf(U[] original, int newLength,
                               Class<? extends T[]> newType);
```

Mit dieser Variante kann man ein Array in ein anderes kompatibles Array kopieren. Kompatible Typen sind alle Arrays von Super-Typen der Elemente des Original-Arrays. Ist der Element-Typ eine Klasse sind dies die Superklassen oder die von der Klasse implementierten Interfaces. Ist der Element-Typ ein Interface, sind es die Super-Interfaces (oder `Object`). Der Typ des neuen Arrays wird mit Hilfe der zugehörigen `Class`-Instanz als letztes Argument übergeben. Damit ist dann auch eine Überprüfung möglich, aber leider erst zur Laufzeit. Unabhängig davon, ob man nun Generics versteht oder nicht, können generische Methoden wie normale Methoden aufgerufen werden. Lässt man alle spitzen Klammer wie `<T,U>` einfach weg und ersetzt `T` und `U` durch `Object`, wird die Methode sogar recht einfach. Schließlich gibt es noch eine hilfreiche Ergänzung zu `copyOf()`:

```
T[] copyOfRange(T[] originalArray, int from, int to);
```

Die Methode `copyOfRange()` besitzt anstatt der neuen Länge zwei Parameter, um den von-bis-Kopierbereich festzulegen. Die Kopie startet bei `from` und hört bei `to-1` auf (also exklusiv `to`). Somit ist die Länge des neuen Arrays `to-from`. Ein Beispiel.

Listing 1.5 Array-Kopien mittels copyOf() und copyOfRange()

```
public class TestArrayCopy {

  public static void main(String... args) {
    int[] iArr= {1,2,3};
    int[] jArr= new int[6];
    int[] kArr= new int[2];

    // --- bis Java 5: die alte Art, Arrays zu kopieren
    System.arraycopy(iArr,0,jArr,0,iArr.length);
    System.arraycopy(iArr,0,kArr,0,kArr.length);

    // --- ab Java 6: die neue Art, Arrays zu kopieren
    jArr= Arrays.copyOf(iArr,6);
    kArr= Arrays.copyOf(iArr,2);
    System.out.println(Arrays.toString(jArr)); ➪ [1, 2, 3, 0, 0, 0]
    System.out.println(Arrays.toString(kArr)); ➪ [1, 2]
```

[9] Generics werden erst im nächsten Kapitel behandelt.

```
      // --- Kopieren eines Teils eines Arrays
      Double[] dArr= {1.,2.,3.};
      System.out.println(Arrays.toString(
         Arrays.copyOfRange(dArr, 0, 2)));        ➪ [1.0, 2.0]

      // --- Kopieren in ein kompatibles Array:

      // ---  Number ist ein Super-Typ vom Element-Typ Double des Originals dArr
      Number[] nArr= Arrays.copyOf(dArr,6,Number[].class);

      // ---  Object ist ein Super-Typ vom Element-Typ Double des Originals dArr
      Object[] oArr= Arrays.copyOf(dArr,6,Object[].class);

      System.out.println(dArr.getClass()); ➪ class [Ljava.lang.Double;
      System.out.println(nArr.getClass()); ➪ class [Ljava.lang.Number;
      System.out.println(oArr.getClass()); ➪ class [Ljava.lang.Object;

     // --- Mit der class-Angabe wird erzwungen, dass nun oArr ein Number-Array
     //     referenziert. Das ist möglich, aber sicherlich verwirrend:
      oArr= Arrays.copyOf(dArr,6,Number[].class);

      System.out.println(oArr.getClass()); ➪ class [Ljava.lang.Number;

      // --- Das kompiliert zwar, aber führt in beiden Fällen zur Ausnahmen, denn:
      //     Double ist keine Superklasse von Float und Float nicht von Double
      Float[] fArr= {1f,2f,3f};

      dArr= Arrays.copyOf(fArr,6,Double[].class);
          ➪ Exception in thread "main" java.lang.ArrayStoreException

      fArr= Arrays.copyOf(dArr,6,Float[].class);
          ➪ Exception in thread "main" java.lang.ArrayStoreException
    }
}
```

Die letzten beiden Anweisungen werden leider vom Compiler nicht als fehlerhaft erkannt![10] Dagegen können einfache Unverträglichkeiten wie beispielsweise

```
nArr= Arrays.copyOf(dArr,6,Object[].class);
```

durchaus vom Compiler erkannt werden. Hier soll ein `Object`-Array erschaffen werden, aber einem `Number`-Array zugewiesen werden. Das ist bereits statisch ein Fehler.

[10] Voraus eilende Generics-Erklärung: `T` muss ein Supertyp von `U` sein. Aber `T` und `U` haben in `copyOf()` keine logische Abhängigkeit. Folglich kommt es bei einem Typ `T`, der kein Supertyp von `U` ist, zu einer `ArrayStoreException` zur Laufzeit.

1.11 Erweiterung des Collection-APIs

Für die üblichen Programmieraufgaben sind Kollektionen und Arrays wichtiger als Sprachерweiterungen. Bereits in Java 5 wurde das Collection-Framework stark erweitert. Es kam insbesondere ein neues Package `java.util.concurrent` hinzu, dass echte Parallelverarbeitung in Kollektionen zulässt und zusätzlich sogar ein solides Fundament für asynchrone Programmierung bietet. Ohne das Concurrent-Package war man bis dato gezwungen, mit Hilfe `Vector`, `HashTable`, synchronized Wrapper – abgesehen von eigenen Implementierungen – nicht wirklich performante Container zu benutzen. Nun gibt es attraktive Alternativen. Die Ergänzungen in Java 6 sehen zwar wenig spektakulär aus, dokumentieren aber wichtige Detailarbeit. Mit `ConcurrentSkipListSet` bzw. `ConcurrentSkipListMap` wurde sogar eine innovative Implementierung eines *lock-free* Algorithmus vorgestellt. Nachfolgend beschränken wir uns auf die Java 6-Erweiterungen. Die bereits vorhandenen Java 5 Container werden allerdings im Kontext mit besprochen.

1.11.1 API-Überblick

Aus der Sicht von Datenstrukturen bietet dass API vier unterschiedliche Container-Typen: *Menge* (Set), *Liste* (List), *Warteschlange* (Queue) und *assoziatives Array*[11] (Map). Eine kompakte Übersicht unter Weglassen der generischen Parameter bietet die folgende tabellarische Übersicht.[12]

Interfaces	***nicht-abstrakte Implementierungen***
`Set`	`HashSet, EnumSet, LinkedHashSet, CopyOnWriteArraySet`$_c$
`SortedSet`	
`NavigableSet`	`TreeSet, ConcurrentSkipListSet`$_c$
`List`	`ArrayList, LinkedList, CopyOnWriteArrayList`$_c$
`Queue`	`PriorityQueue`
`BlockingQueue`	`ArrayBlockingQueue`$_c$`, DelayQueue`$_c$`, LinkedBlockingQueue`$_c$`, PriorityBlockingQueue`$_c$`, SynchronousQueue`$_c$
`Deque`	`LinkedList, ArrayDeque`
`BlockingDeque`	`LinkedBlockingDeque`$_c$

[11] Kann man über einen Schlüsseln und nicht (nur) über die Position auf ein Array-Element zugreifen, nennt man das Array assoziativ.

[12] Das `'c'` im Index bedeutet: gehört zu Package `java.util.concurrent`.Veraltete Klassen wie `Vector`, `Stack` oder `HashTable` wurden weggelassen, da es bessere Concurrent-Alternativen gibt.

Interfaces	*nicht-abstrakte Implementierungen*
Map	HashMap, LinkedHashMap, IdentityHashMap, ENumMap, WeakHashMap
SortedMap	
ConcurrentMap	ConcurrentHashMap$_c$
NavigableMap	TreeMap
Concurrent-NavigableMap	ConcurrentSkipListMap$_c$

Zu den beiden Interfaces `SortedSet` und `SortedMap` gibt es keine direkte Implementierung. Queues, die erst ab Java 5 eingeführt wurden, haben dagegen viele unterschiedliche Implementierungen. Es gibt zwei Interface-Hierarchien (Abbildung 1.4 bzw. 1.5), wobei in zwei Fällen auch die Mehrfach-Typ-Vererbung von Interfaces genutzt wird.

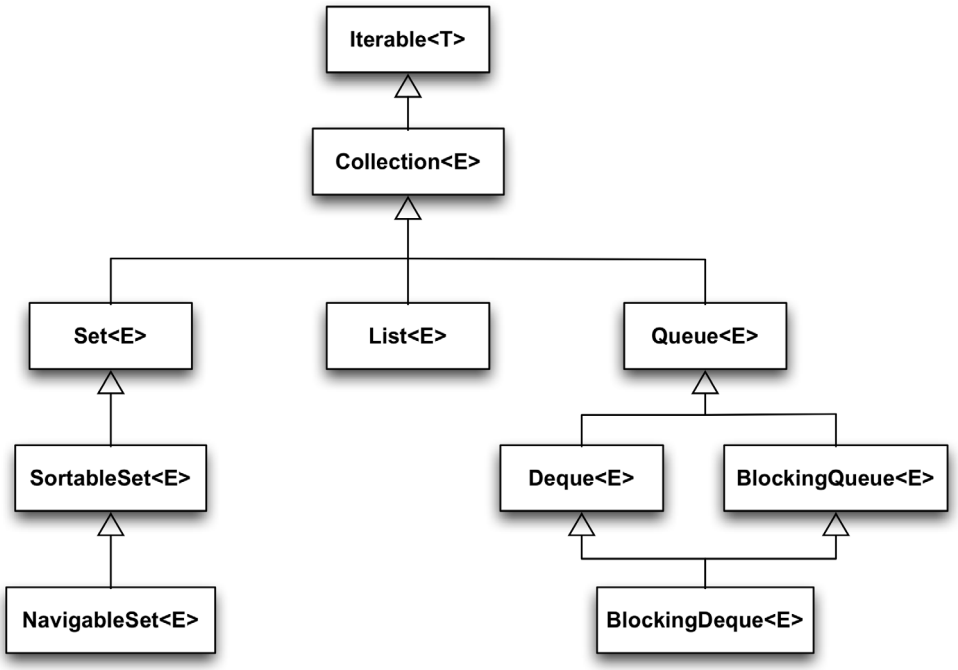

Abbildung 1.4 Collection-Hierarchie

Für Kollektionen ist der Ausgangspunkt seit Java 5 ein `Iterable`-Interface. Es wurde hinzugefügt, um das Iterieren aller Kollektionen mittels der for-each Schleife zu ermöglichen.

Die Blocking-Interfaces befinden sich im Package `java.util.concurrent`. Sie machen nur im Zusammenhang mit asynchroner Programmierung Sinn.

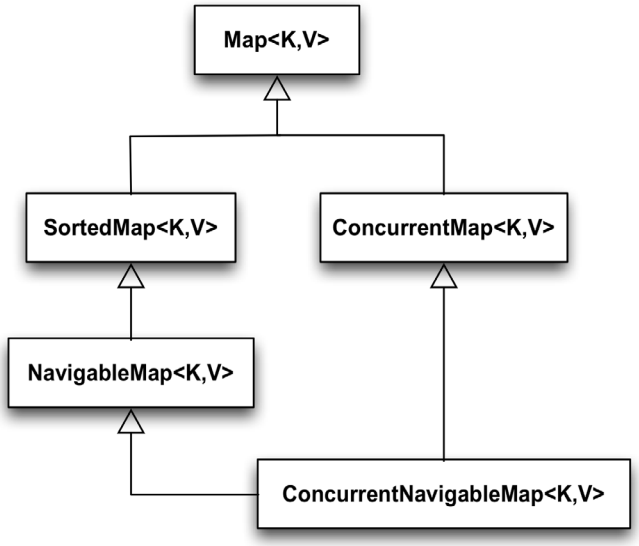

Abbildung 1.5 Map-Hierarchie

Aufgrund der drei Map-Methoden

- `Set<Map.Entry<K,V>> entrySet()`
- `Set<K> keySet()`
- `Collection<V> values()`

können Maps immer unter der Sicht einer Kollektion betrachtet werden. Die beiden Concurrent-Map-Interfaces gehören zum Package `java.util.concurrent`. Beide Hierarchien wurden in Java 6 um Navigation erweitert. Wie der Name besagt, erlaubt dies ein auf- oder absteigendes Durchlaufen des Containers. Interessant ist die Namensgebung. Sie steht im Zusammenhang mit den folgenden Begriffen:

- **Bounded**
  Einen Container nennt man gebunden, wenn seine Aufnahmekapazität für Elemente limitiert werden kann. Unbounded Container sind unbeschränkt wachsend.
- **Blocking**
  Ist für eine Operation die entsprechende Bedingung (Precondition) nicht erfüllt, kann die Operation so lange suspendiert (blockiert) werden, bis die Bedingung gegeben ist. Das macht nur Sinn bei asynchroner Programmierung, beispielsweise weil die Kapazität eines bounded Container beim Einfügen überschritten wird oder kein Element zur Entnahme existiert.

- **Concurrent**
  Dieses Präfix signalisiert den sicheren Zugriff auf den Container von mehreren Threads aus. Die so genannten synchronized Wrapper bieten zwar einen ähnlichen Service, aber nur die Concurrrent-Versionen sind wirklich performant.
- **Navigable**
  Im Gegensatz zu den sortierten Vorgängern kann ein navigierbarer Container vorwärts und rückwärts traversiert werden und erlaubt die Suche nach nächstliegenden (*closest match*) Elementen.
- **Ordering**
  Die *natürliche* Ordnung (natural ordering) wird durch das Interface `Comparable` festgelegt. Für eine abweichende Ordnung muss man bei der Anlage einen `Comparator` übergeben. Zusätzlich besteht bei der Anlage des Containers noch die Möglichkeit, einen anderen sortierten zu übergeben, dessen Ordnung übernommen werden soll (siehe auch Hinweis 1.9 mit Beispiel).

Im Gegensatz zur ersten Tabelle oben werden in der nachfolgenden Tabelle nur die in Java 6 neu hinzugekommen Interfaces mit ihren zugehörigen Implementierungen vorgestellt.

Interface	*Direkte Implementierung*
`NavigableSet<E>`	**`TreeSet<E>`** : Bereits vorhandene nicht thread-sichere Klasse, retrofitted.[13]
	`ConcurrentSkipListSet<E>` : Basiert auf `ConcurrentSkipListMap`.
`NavigableMap<K,V>`	**`TreeMap<K,E>`** : Bereits vorhandene nicht thread-sichere Klasse, retrofitted.
	`ConcurrentSkipListMap<K,V>` : Neue Klasse, basiert auf einem lock-free *CAS*-Algorithmus. Ein Compare-And-Set-Algorithmus verwendet atomare Operationen der Prozessoren und keine Locks und ist daher sehr effizient.
`Deque<E>`	**`LinkedList<E>`** : Bereits vorhandene nicht thread-sichere Klasse, retrofitted.
	`ArrayDeque<E>` : Neue nicht thread-sichere Klasse, die wie `ArrayList` array-basiert ist.
`BlockingDeque<E>`	**`LinkedBlockingDeque<E>`** : Thread-sichere verkettete Liste, optional mit Angabe der Bound (sonst `Integer.MAX_VALUE`).

Die in der Tabelle aufgeführten Interfaces und ihre zugehörigen Implementierungen werden im nächsten Abschnitt anhand von Beispielen vorgestellt.

[13] Zu retrofitted siehe Unterabschnitt in 1.3.1

1.11.2 Utility-Klasse Collections

Die Klasse `Collections` ist ein Sammelbecken für Algorithmen bzw. statische Methoden, denen verschiedene Kollektions-Typen zugeordnet sind. Mit wenigen Ausnahmen ist deshalb zumindest der erste Parameter eine Kollektion. Auch diese Klasse wurde um zwei Utility-Methoden erweitert.

asLifoQueue() alias Stack

Da es bereits eine Klasse `Stack<E>` gibt, ist dies wohl der Grund, dass es kein Interface mit diesem Namen gibt, was sicherlich eine saubere Typ-Lösung gewesen wäre. Leider ist aber die Klasse `Stack<E>` einem Design-Fehler zum Opfer gefallen und von `Vector<E>` abgeleitet. Somit erbt sie zusätzlich alle Methoden von `Vector`. Beide sind zwar retrofitted, aber „sub-optimal“ implementiert.

In Java 6 gibt es deshalb eine neue Methode `asLifoQueue()`, die eine `Queue` liefert, die sich wie ein Stack verhält. Das ist vom Ansatz her recht merkwürdig. Denn der Rückgabetyp signalisiert FIFO, der Methoden-Name dagegen einen LIFO alias Stack. Damit das funktioniert, muss man eine `Deque`-Instanz übergeben. Intern werden dann alle `Queue`-Methoden, die stack-konform sind, auf die `Deque`-Methoden „umgebogen“. Somit wird die `Queue`-Methode `add()` logisch zu `push()` und `remove()` zu `pop()`. Das ist nicht so gut wie ein Stack-Interface, verhindert aber zumindest Fehler, wenn man einen Stack mittels `Deque` simulieren will und ist im Gegensatz zu `Stack` performant. Ein Beispiel:

```
Queue<String> stack= Collections.asLifoQueue(
                                 new ArrayDeque<String>());
stack.add("A");
stack.add("B");
stack.addAll(Arrays.asList("C","D","E"));
System.out.println(stack);                  ➪ [E, D, C, B, A]

// --- Bedingung genügt!
for (;!stack.isEmpty();)
  System.out.print(stack.remove()+" ");     ➪ E D C B A
```

Die Methode `addAll()` kann natürlich nicht einfach intern die gleiche Methode von `Queue` aufrufen, sondern musste neu implementiert werden.

newSetFromMap()

Da Sets mit Hilfe von Maps implementiert werden können, war es nur logisch, eine neue Methode `newSetFromMap()` anzubieten, der man eine Map als Basis für eine Set übergeben kann. Der Vorteil: Eine Set übernimmt dann alle Eigenschaften wie die Ordnung oder die Thread-Sicherheit von einer Map.

Anhand des zweiten Typ-Arguments `Boolean` in

```
public static <E> Set<E> newSetFromMap(Map<E,Boolean> map)
```

erkennt man, dass das `Set`-Element als Schlüssel mit genau einem zugehörigen Wert `Boolean.TRUE` oder `Boolean.FALSE` in der Map eingetragen wird. Genau das implementiert eine Menge. Wichtig dabei ist, dass die Map, die der `newSetFromMap()` übergeben wird, leer ist. Die Methode sollte auch nicht für Maps benutzt werden, die bereits als Basis für vorhandene Set-Implementierungen dienen. Dazu gehören:

- `HashSet`, basiert auf `HashMap`
- `TreeSet`, basiert auf `TreeMap`
- `ConcurrentSkipListSet`, basiert auf `ConcurrentSkipListMap`

Sinnvoller sind beispielsweise folgende Sets:

```
Set<String> idSet= Collections.newSetFromMap(
                          new IdentityHashMap<String,Boolean>());
Set<String> lnkSet= Collections.newSetFromMap(
                          new LinkedHashMap<String,Boolean>());

String a= new String("a");
idSet.addAll(Arrays.asList("a","b",a,"a"));
lnkSet.addAll(Arrays.asList("a","b",a,"a"));
System.out.println(idSet);                    ➪ [a, a, b]
System.out.println(lnkSet);                   ➪ [a, b]
```

Da der Wert hinter der Variablen `a` und das Literal `"a"` unterschiedliche Instanzen sind, werden in der `IdentityHashMap` beide als unterschiedliche Elemente aufgenommen. Die beiden Literale `"a"` sind allerdings identisch. Die `LinkedHashMap` fügt dagegen aufgrund des Wertvergleichs nur ein `"a"`-Element ein.

Thread-sichere Collections-Wrapper

Mit Ausnahme von *deprecated*[14] Klassen wie `Vector` sind alle Kollektionen im Package `java.util` nicht thread-sicher implementiert. Braucht man Thread-Sicherheit, besteht eine Möglichkeit darin, zur der jeweiligen Implementierung eine entsprechende thread-sichere Variante zu bekommen. Dazu greift man dann auf die Utility-Klasse `Collections` zurück. Sie enthält statische Hilfsmethoden, die thread-sichere Wrapper für Kollektionen liefern. Das ist weder performant noch objektorientiert. Der erste Nachteil von Wrappern liegt darin, dass sie nur das kapseln, was sie auch kennen. Sie passen sich nicht automatisch an neue Klassen oder Interfaces mit neuen Methoden an. Das erfordert entweder einen neuen Wrapper oder die Anpassung des alten, da Wrapper Methoden nicht automatisch erben!

[14] deprecated: missbilligt, d.h. ersetzt und verbessert durch Neueres.

Beispielsweise gibt es für die neuen navigierbaren Sets zur Zeit keine passenden Wrapper. Eine navigierbare Set wird also per Wrapper in eine thread-sichere sortierte Set umwandeln:

```
// --- synchronizedNavigableSet() gibt es nicht!
SortedSet<String> ss= Collections.synchronizedSortedSet(navSet);
```

Das ist nicht unbedingt ein Nachteil! Das zweite Handicap von Wrappern ist dagegen gravierender als das erste. Per `synchronized` (*intrinsic* locks[15]) werden parallele Zugriffe auf die jeweilige Kollektion ausgeschlossen. An sich setzt man Threads ja gerade dann ein, wenn man parallele Bearbeitung forcieren möchte. Sind praktisch alle Methoden synchronisiert, kann man genau so gut – analog zu einem Event-Dispatcher bei der GUI – nur mit einer Thread arbeiten. Anstatt neue Wrapper zu kreieren, ist es demnach besser, unmissverständlich auf das Concurrent-Package hinzuweisen. Denn für echte Parallel-Bearbeitung sollte man die Container in `java.util.concurrent` nutzen.

1.11.3 Queue, Deque und ConcurrentStack

In Java 6 werden Queues um eine wichtige Variante erweitert. Eine *Deque*[16] ist eine sogenannte „double ended Queue". Bei einer normalen Warteschlange werden nur am Ende Elemente eingefügt und am Kopf entnommen, bei einer Deque von beiden Seiten. Das Interface `Deque` wird im Package `java.util` vom Interface `Queue` abgeleitet. `Deque` hat neben `LinkedList` eine neue nicht thread-sichere direkte Implementierung `ArrayDeque` sowie eine thread-sichere Klasse `LinkedBlockingDeque`, die direkt `BlockingDeque` – ein Sub-Interface von `Deque` – implementiert.

Man unterscheidet Deques mit und ohne Limitierung der Aufnahmekapazität. `ArrayDeque` und `LinkedList` nehmen unbeschränkt viele Elemente auf, bei `BlockingQueue` kann dagegen die Anzahl limitiert werden. Wie auch aus der ersten Tabelle zu ersehen, implementiert die Klasse `LinkedList` drei Interfaces: `List`, `Queue` und `Deque`. Sie sollte daher nicht mehr als eigene Typen auftreten. Sonst mischt man munter unterschiedliche Methoden (siehe hierzu auch Abschnitt 1.3). Von einem klaren Programmierstil erwartet man eher eine `List`, eine `Queue` oder `Deque`, nicht aber eine Melange aus allen dreien.

```
// --- anstatt: LinkedList deque= new LinkedList();
Deque deque= new LinkedList();
```

Benötigt man eine `Deque`, hilft die IDE und der Compiler, verwendet man dagegen die auskommentierte Variante, wird man mit einem grundlegenden Ingenieur-Prinzip konfrontiert.[17]

Beim Einfügen, Löschen und Betrachten von Elementen einer `Deque` gibt es jeweils zwei verschiedene Varianten, sofern die Operation nicht durchgeführt werden kann. Die eine Va-

---

15 Lock-Mechanismus, der bei Java in jedem Objekt eingebaut ist.

16 Deque wird wie Deck ausgesprochen!

17 „Alles, was schief gehen kann, geht auch schief!" Jeder Informatiker lernt das und umschreibt *that-which-shall-not-be-named* passend mit: „Der Rechner hat einen Virus", „OS/Version/Plattform ist inkompatibel!"

riante löst eine Ausnahme aus und die andere liefert beim Löschen oder Betrachten `null` bzw. beim Einfügen `false`. Für Deques, die nur eine beschränkte Anzahl von Elemente zulassen, ist wohl die letzte Variante besser geeignet. Damit gibt es zwölf spezielle `Deque`-Methoden, wobei sechs in der Wirkung absolut gleich mit der `Queue` sind.[18]

Op	*Typ*	*Am Kopf*		*Am Ende*	
		Ausnahme	*null / false*	*Ausnahme*	*null / false*
E	*Deque*	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
	Queue	-	-	`add(e)`	`offer(e)`
	Stack	`push(e)`	-	-	-
L	*Deque*	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
	Queue	`remove()`	`poll()`	-	-
	Stack	`pop()`	-	-	-
B	*Deque*	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`
	Queue	`element()`	-	-	-
	Stack	`peek()`	-	-	-

Da `null` als spezieller Wert für nicht-vorhandene Elemente benutzt wird, kann `null` nicht in eine `Deque` eingefügt werden. Ein Versuch wird mit einer `NullPointerException` bestraft. Speziell für eine `BlockingQueue` oder `BlockingDeque` gibt es für Einfüge- und Lösch-Operationen noch zwei weitere Varianten.

Operation		*Ausnahme*	*null / false*	*Blockierend*	*Zeitschranke t=timeout, u=unit*
E	*Kopf*	`addFirst(e)`	`offerFirst(e)`	`putLast(e)`	`offerFirst(e,t,u)`
	Ende (Queue)	`addLast(e)` `add(e)`	`offerLast(e)` `offer(e)`	`putLast(e)` `put(e)`	`offerLast(e,t,u)` `offer(e,t,u)`
L	*Kopf (Queue)*	`removeFirst()` `remove()`	`pollFirst()` `poll()`	`takeLast()` `take()`	`takeFirst(t,u)` `take(t,u)`
	Ende	`removeLast()`	`pollLast()`	`takelast()`	`takeLast(t,u)`

[18] In der ersten Spalte steht *E* dabei für Einfügen, *L* für Löschen und *B* für Betrachten.

Sun wäre nicht Sun, wenn es nicht gegen die reine Lehre der Algorithmen & Datenstrukturen verstoßen dürfte. Denn `Deque` bietet zwei Methoden, die Elemente innerhalb der `Deque` zu löschen. Da die beiden Methoden wohl häufig auch für Deques benutzt werden, wäre die Alternative gewesen, eine `List` anstatt einer `Deque` zu nehmen. Angesichts des o.a. Ingenieurs-Prinzips ist das (für Sun) nicht wirklich attraktiv.[19]

```
removeFirstOccurrence(Object o);    // Löscht erstes Auftreten von o
removeLastOccurrence (Object o);    // Löscht letztes Auftreten von o
```

Deque

Da Queues bereits seit Java 5 existieren, werden im Folgenden einige Methoden aus dem `Deque` Interface am Beispiel vorgestellt.

Listing 1.6 (Nicht-) blockierende Deque

```
public class TestDeque {
   public static void main(String... args) {

      // --- die nicht-blockierende Variante ohne Kapazitätsbeschränkung:
      //     Start mit 8 Elementen, ohne Angabe ist der Default 16
      Deque<String> dq= new ArrayDeque<String>(8);

      // --- nur holen, nicht entfernen: null= wenn leer
      System.out.println(dq.peekFirst());                    ➪ null
      System.out.println(dq.peekLast());                     ➪ null

      // --- Einfügen:
      //     Wenn nicht möglich: IllegalStateException bei der add-Variante
      //                         false bei der offer-Variante
      dq.addFirst("Hallo");
      System.out.println(dq.offerFirst("Hallo"));            ➪ true
      System.out.println(dq.offerLast("Welt");               ➪ true
      System.out.println(dq);                  ➪ [Hallo, Hallo, Welt]

      // --- poll-Methoden: holen und entfernen: null= wenn leer
      System.out.println(dq.pollFirst());                    ➪ Hallo
      dq.offerFirst("Welt");

      // --- remove(First/Last):  holen und entfernen. Ausnahme, wenn leer
      //     remove(First/Last)Occurence:  false= wenn nicht möglich
      System.out.println(dq.removeLastOccurrence("Welt"));   ➪ true
      System.out.println(dq);                          ➪ [Welt, Hallo]
```

[19] Vielleicht wäre ein Sub-Typ `SunDeque` origineller gewesen.

```
    // --- die blockierende Variante:
    //     Kapazität 2, ohne Angabe ist der Default: Integer.MAX_INT
    BlockingDeque<String> bdq=
                          new LinkedBlockingDeque<String>(2);

    System.out.println(bdq.offerFirst("Hallo"));          ➪ true
    System.out.println(bdq.offerFirst("Hallo"));          ➪ true
    System.out.println(bdq.offerLast("Welt"));            ➪ false
    System.out.println(bdq.pollLast());                   ➪ Hallo
    System.out.println(bdq.pollFirst());                  ➪ Hallo
    System.out.println(bdq.pollLast());                   ➪ null
    bdq.addFirst(null);                  ➪ ... NullPointerException
  }
}
```

Die letzte Anweisung löst eine `NullPointerException` aus, da der Wert `null` als Element nicht erlaubt ist.

CAS-Implementierung: ConcurrentStack

Die Stack-Lösung mit `Collections.asLifoQueue()` ist zwar pragmatisch, jedoch logisch nicht zufriedenstellend. Brian Goetz hat 2006 mit Hilfe eines nicht-blockierenden *CAS*-Algorithmus eine Implementierung des Stacks vorgestellt. CAS steht für eine atomare *Compare-And-Set* Operation und kann somit in speziellen Fällen `synchronized` ersetzen. Zusammen mit atomaren Referenzen ist die Implementierung eines Stacks erstaunlich kurz. Dies ist auch einer der beiden Gründe der Vorstellung. Der andere liegt darin, dass die Implementierung einen guten Eindruck von der Wirkungsweise von CAS und atomaren Referenzen vermittelt. Das geht zwar über die reine Nutzung von Kollektionen hinaus, ist aber für ein besseres Verständnis der Container sehr hilfreich.

Obwohl der Name Stack als Klasse bereits vergeben ist, verwenden wir im Folgenden aus Klarheit ein Interface `Stack`:

```
// --- die essentiellen Stack-Methoden
public interface Stack<E> {
  boolean empty();
  E peek();
  E pop();
  void push(E element);
}
```

Die folgende CAS-Implementierung ist zwar generisch konzipiert, aber auch ohne tiefe generische Kenntnisse verständlich.

Listing 1.7 Eine ConcurrentStack-Implementierung

```
public class ConcurrentStack<E> implements Stack<E> {
  // --- Atomare Referenzen: wie volatile Variable, aber mit thread-sicherem Update
  private final AtomicReference<Node<E>> top=
                            new AtomicReference<Node<E>>();

  private static class Node<E> {
    // --- thread-sichere Variable, nur in ConcurrentStack im Zugriff
    private final E element;
    private final Node<E> next;

    // --- aufgrund von finals thread-sichere Anlage
    private Node(E element, Node<E> next) {
      this.element= element; this.next= next;
    }
  }

  // --- atomicVariable.get() wie volatile thread-sicher
  public boolean empty() {
    return top.get()==null;
  }

  public void push(E element) {
    Node<E> curTop;
    // --- siehe hierzu Kommentar nach Listing!
    while (!top.compareAndSet(curTop=top.get(),
                              new Node<E>(element,curTop)));
  }

  public E pop() {
    Node<E> curTop;
    do {
      curTop= top.get();
      if(curTop==null)
        return null;
    } while (!top.compareAndSet(curTop,curTop.next));
    return curTop.element;
  }

  public E peek() {
    Node<E> curTop;
    return (curTop=top.get())==null?null:curTop.element;
  }
}
```

Die statisch innere Klasse `Node` ist rein `private`, da sie nur innerhalb von `ConcurrentStack` benutzt werden soll. Der Kern der `push`-Operation ist CAS:

```
top.compareAndSet(curTop=top.get(),new Node<E>(element,curTop));
```

Die Argumente in der Methode werden vor dem Aufruf zuerst von links nach rechts ausgewertet. Aufgrund der atomaren Referenz `top` wird auch `curTop` atomar bestimmt. Dann wird eine neue `Node`-Instanz erschaffen. Konstruktor zusammen mit `private final` gewährleisten Thread-Sicherheit. Bei dem nun folgenden CAS-Aufruf wird `curTop` erneut mit dem aktuellen Top verglichen, der sich zwischenzeitlich verändert haben kann. Nur wenn der `curTop` noch identisch zum aktuellen Top ist, erfolgt atomar die nachfolgende set-Operation mit der `Node`-Instanz und das Ergebnis ist `true`. Ist dies nicht der Fall und `compareAndSet()` liefert `false`, wird die `while`-Schleife erneut durchlaufen.

Gegenüber den Algorithmen, die auf Locks basieren, ist CAS dann performanter, wenn eine Modifikation zwischen Holen und Setzen eines Werts durch eine andere Thread eher selten ist. Das ist in den weitaus meisten Fällen auch der Fall. CAS ist somit eine Art von optimistischen Locking, wogegen Blocking-Algorithmen eine pessimistische Strategie verfolgen.

Die `pop`-Operation läuft logisch identisch zu der `push`-Operation ab. Aufgrund der Design-Entscheidung `null` zu liefern, sofern der Stack leer ist, wurde eine `do-while` gewählt. In peek() wie in den anderen beiden Methoden ist `curTop` eine lokale Variable, also thread-sicher.

1.11.4 Konsistenz von hashCode(), equals() und compare()

Mit Ausnahme einer so genannten Identitäts-Menge wie beispielsweise

```
Collections.newSetFromMap(new IdentityHashMap<String,Boolean>());
```

ist für Sets der Wertvergleich beim Einfügen neuer Elemente essentiell. Die von der Klasse `Object` geerbte Methode `equals()` prüft prinzipiell nur auf Identität. Jede Klasse, die ihre Instanzen anhand ihrer Werte vergleichen möchte, muss also einen so genannten *strukturellen* Vergleich durchführen.

Für viele mathematischen Objekte ist dies einfach: Nur wenn aller ihre Felder gleich sind, sind auch zwei Instanzen gleich. Nicht-mathematische Objekte haben dagegen oft künstliche Schlüssel[20] zur Identifikation und die prüft `equals()` dann auf Gleichheit. Bei sortierten Containern ist neben der Gleichheit noch die Ordnung entscheidend. Denn nur total geordnete Instanzen einer Klasse lassen sich sortieren. Zwischen der Methode `hashCode()` und `equals()` sowie zwischen `equals()` und `compare()` gibt es Abhängigkeiten, die im weiteren anhand von Beispielen demonstriert werden.

[20] Wie die Matrikel-Nummer für Studierende oder die RFIDs für die automatischen Identifizierung.

Zuerst benötigen wir eine möglichst einfache Klasse wie `Point`. In der ersten Version soll sie nur die Methode `equals()` enthalten.

```
public class Point {
  protected int x,y;

  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }

  @Override
  public String toString() {
    return "("+x+","+y+")";
  }

  @Override
  public boolean equals(Object o) {

    // --- wird nur optional für Nachweis des Aufrufs benötigt!
    System.out.println("equals");
    return o instanceof Point?
           x==((Point)o).x && y==((Point)o).y : false;
  }
}
```

Führen wir einen kurzen Test durch:

```
Set<Point> pSet1= new HashSet<Point>();
Point p1= new Point(1,1);
Point p2= new Point(1,2);

System.out.println(pSet1.add(p1));          ➪ true
System.out.println(pSet1.add(p2));          ➪ true
System.out.println(pSet1);                  ➪ [(1,2), (1,1)]

Point p= new Point(1,1);
System.out.println(pSet1.contains(p));      ➪ false
```

Laut Set-Kontrakt (siehe hierzu die Java-Dokumentation) testet die Methode `contains()` mittels `equals()`, ob eine Instanz in der Menge ist. Da sich `equals()` beim Aufruf melden müsste, wird `equals()` offensichtlich gar nicht erst aufgerufen und das Ergebnis von `contains()` ist `false`. Das ist logisch nicht haltbar, liegt aber an dem Kontrakt zwischen `hashCode()` und `equals()`. Deshalb fügt man im zweiten Schritt eine entsprechende Methode `hashCode()` zur Klasse `Point` hinzu:

```
public class Point {
  // --- wie oben ...

  @Override
  public int hashCode() {
    return x^y;
  }
}
```

Nun liefert die letzte Anweisung im Test oben das erwünschte Ergebnis:

```
System.out.println(pSet1.contains(p));      ➪ equals
                                            ➪ true
```

Für keine Klasse, die man implementiert, kann man ausschließen, dass ihre Instanzen in Containern wie beispielsweise `HashSet` oder `LinkedHashSet` eingefügt werden. Deshalb sollte der erste Punkt des folgenden Hinweises immer beachtet werden.

Hinweis 1.9 equals, hashCode und compare(To)

- Damit `equals()` immer aufgerufen wird, muss das Resultat von `hashCode()` für Instanzen gleich sein, deren `equals()`-Vergleich `true` liefert. Die Umkehrung gilt nicht!
- Implementiert eine Klasse das Interface `Comparable` mit der Methode `int compareTo(T o)`, sagt man, die Klasse ist *natürlich* geordnet.
- Benötigt man eine von der natürlichen Ordnung abweichende Sortierung bzw. ist die Klasse nicht natürlich geordnet, kann man nachträglich mit Hilfe des Interfaces `Comparator` eine externe Methode `int compare(T o1,T o2)` angeben.

`T` steht wie `E` bei Kollektionen für einen beliebigen Typ, der durch die aktuelle Klasse ersetzt wird. Das Interface `Comparator` wird häufig als anonyme Klasse implementiert.

Um im Folgenden die Beziehung zwischen `equals()` und `compare()` zu testen, erweitern wir die `Point`-Klasse um zwei Distanz-Methoden. Die erste Methode `distance()` beruht auf dem euklidischen Abstand zum Ursprung. Eine weitere Methode `blockDistance()` – in USA auch *Manhattan-Distance* genannt – bestimmt den Abstand zum Ursprung entlang der x,y-Achsen. Der Block-Abstand ist somit der normale Weg, den man in amerikanischen Städten wie Manhattan zwischen zwei Punkten zurücklegen muss.

Für die Klasse `Point` führen wir nun eine natürliche Ordnung auf Basis der Block-Distanz ein. Dazu muss `Point` das Interface `Comparable<Point>` mit der einzigen Methode `compareTo()` implementieren. Der größere Abstand zum Ursprung entscheidet beim Vergleich zweier Punkte, welcher größer ist.

Listing 1.8 Eine Point-Klasse mit zwei Ordnungen

```
public class Point implements Comparable<Point> {
  public static final Point ORIGIN= new Point(0,0);    // Ursprung
  protected int x,y;

  public Point() {}

  public Point(int x,int y) {
    this.x=x; this.y=y;
  }

  @Override
  public int hashCode() {
    return x^y;
  }

  @Override
  public String toString() {
    return "("+x+","+y+")";
  }

  // --- Vergleich der x,y-Werte. Sie ist unabhängig von Abstands-Messungen!
  @Override
  public boolean equals(Object o) {
    return o instanceof Point? x==((Point)o).x && y==((Point)o).y
                             : false;
  }

  // --- 1. Variante zum  Abstand zweier Punkte
  public int blockDistance(Point p) {
    return Math.abs(x-p.x)+Math.abs(y-p.y);
  }

  // --- 2. Variante zum  Abstand zweier Punkte
  public double euclidianDistance(Point p) {
    return Math.sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
  }

  // --- Ein Punkt ist größer als ein anderer, wenn er weiter vom Ursprung entfernt ist.
  public int compareTo(Point p) {
    return blockDistance(ORIGIN)==p.blockDistance(ORIGIN)?0:
           blockDistance(ORIGIN)>p.blockDistance(ORIGIN)?1:-1;
  }
}
```

Der nächste Test verwendet die Klasse `TreeSet`. Hier gibt es eine böse Überraschung, die auf die Inkonsistenz von `equals()` und `compareTo()` zurückzuführen ist:

```
Set<Point> pSet1= new TreeSet<Point>();

// --- zwei verschiedene Punkte
Point p1= new Point(-2,-2);
Point p2= new Point(0,4);

pSet1.add(p1);
pSet1.add(p2);
System.out.println(pSet1);                          ➪ [(-2,-2)]
System.out.println(pSet1.contains(p2));             ➪ true

// --- import static kap01.Point.*;
System.out.println(p1.blockDistance(ORIGIN));       ➪ 4
System.out.println(p2.blockDistance(ORIGIN));       ➪ 4
```

Die Klasse `TreeSet` ignoriert `equals()` völlig und verwendet nur `compareTo()`. Da es vier Punkte mit Abstand `1`, acht Punkte mit Abstand `2`, etc. gibt, kann von allen Punkten mit gleichem Abstand in eine `TreeSet` nur jeweils einer aufgenommen werden. Dies schließt eine Übereinstimmung von `equals()` und `compareTo()` aus. Der Abstand definiert mathematisch gesehen nur eine *Quasi-Ordnung*. Egal welche Ordnung man wählt, Punkte lassen sich wie komplexe Zahlen nicht total ordnen:

Aus `p1<=p2` und `p2<=p1` folgt nicht `p1=p2`.

Die Idee, in der Deklaration von `Point` einfach das Interface `Comparable` wegzulassen

```
public class Point { ... }
```

ist keine Lösung. Dann wird zur Laufzeit von `TreeSet` eine `ClassCastException` ausgelöst. Eine Alternative zur natürlichen Ordnung besteht darin, dem Konstruktor von `TreeSet` einen `Comparator` in Form einer anonymen Klasse zu übergeben:

```
Set<Point> pSet1= new TreeSet<Point>();
Set<Point> pSet2= new TreeSet<Point>(
  // --- ein Point-Comparator, definiert als anonyme Klasse
  new Comparator<Point>() {
    public int compare(Point p1,Point p2) {
      return p1.euclidianDistance(ORIGIN)==
             p2.euclidianDistance(ORIGIN)?0:
             p1.euclidianDistance(ORIGIN)>
             p2.euclidianDistance(ORIGIN)?1:-1;
    }
});
```

Ein Test mit den beiden Mengen `pSet1` und `pSet2` zeigt bei der Ausgabe die unterschiedlichen Ordnungen:

```
Point p1= new Point(2,2);
Point p2= new Point(-2,-2);
Point p3= new Point(0,3);
pSet1.add(p1); pSet2.add(p1);
pSet1.add(p2); pSet2.add(p2);
pSet1.add(p3); pSet2.add(p3);

System.out.println(pSet1);                ➪ [(0,3), (2,2)]
System.out.println(pSet2);                ➪ [(2,2), (0,3)]
```

Da `TreeSet` neben `Set` auch `SortedSet` und `NavigableSet` implementiert, hängt die Ausgabe von der Sortierung, d.h. von `compare()` bzw. `compareTo()` ab. In beiden Fällen wird aber der Punkt `p2` ignoriert. Zu beachten ist also:

Hinweis 1.10 Konsistenz von equals() und compare()

- Nur im Fall, dass man in `compare()` bzw. `compareTo()` eine totale Ordnung wählen kann, ist die Konsistenz zu `equals()` gewährleistet. Ist dies nicht möglich, führt die Wahl einer `Set`-Implementierung wie beispielsweise `TreeSet`, die nicht mit `equals()` auf Gleichheit prüft, zu Fehlern.

Neben den beiden vorgestellten Konstruktoren in `TreeSet` gibt es noch zwei weitere:

```
TreeSet()                                 // natürliche Sortierung

TreeSet(Comparator<? super E>  c)         // Sortierung basiert auf Comparator

TreeSet(Collection<? extends E> c)        // Übernahme der Elemente, aber
                                          // immer natürliche Sortierung

TreeSet(SortedSet<E> s)                   // Übernahme der Elemente
                                          // und der Sortierung
```

Die letzten beiden Konstruktoren sind mit Vorsicht zu verwenden, denn:

```
System.out.println(
  new TreeSet<Point>(pSet2));                    ➪ [(0,3), (2,2)]
System.out.println(
  new TreeSet<Point>((SortedSet<Point>)pSet2)); ➪ [(2,2), (0,3)]
```

Die `TreeSet`-Instanz wird in der ersten Anweisung also Typ der Variablen `pSet2`, d.h. als einfache `Set` angesprochen. Daher wird auch der Konstruktor aufgerufen, der nur eine `Collection` erwartet. In der zweiten Anweisung wird dagegen durch den Cast nach `SortedSet` der `TreeSet`-Konstruktor für `SortedSet` aufgerufen.

1.11.5 Navigierbare Container

Basierend auf den Interfaces `SortedSet` bzw. `SortedMap` wurden in Java 6 navigierbare Sets bzw. Maps abgeleitet, die man – neben vielen anderen neuen Operationen – in beiden Richtungen durchlaufen kann. Die neuen Interfaces in Package `java.util` hören somit auf `NavigableSet` und `NavigableMap` und im Package `java.util.concurrent` dann auf `ConcurrentNavigableMap`. Je nach Package stehen für die Interfaces entsprechende Implementierungen bereit.

NavigableSet

Als sortiertes `Set` setzt `NavigableSet` entweder eine natürliche Ordnung voraus oder dem Konstruktor wird ein `Comparator` übergeben, wobei die jeweilige Vergleichsmethode nach Hinweis 1.10 konsistent mit `equals`() sein sollten. Das folgende Listing enthält alle interessanten neuen Methoden von `NavigableSet`:

Listing 1.9 Methoden für NavigableSet

```
package kap01;

import java.util.*;
// --- erspart System bei der Ausgabe, siehe unten
import static java.lang.System.out;

public class TestNavSet {

  public static void main(String... args) {
    NavigableSet<String> ns= new TreeSet<String>();
    ns.addAll(Arrays.asList(
         "1","2","10","ä","ab","a","Abc","ABC","AB","z"));

    // --- aufsteigend sortiert: Vorsicht bei  Zahlen und Umlauten!
    out.println(ns);          ➪ [1, 10, 2, AB, ABC, Abc, a, ab, z, ä]

    // --- kleinstes Element, größer/gleich ö existiert nicht, daher:
    out.println(ns.ceiling("ö"));                  ➪ null

    // --- in diesem Fall liegt über zz das Element ä
    out.println(ns.ceiling("zz"));                 ➪ ä

    // --- größtes Element kleiner/gleich 0 existiert nicht, also:
    out.println(ns.floor("0"));                    ➪ null
    out.println(ns.floor("100"));                  ➪ 10
```

```
    // --- Element strikt größer als 0
    out.println(ns.higher("0"));                    ➪ 1

    // --- Element strikt kleiner als 2
    out.println(ns.lower("2"));                     ➪ 10

    // --- Intervall bis 2 exklusive die Grenze 2
    out.println(ns.headSet("2"));                   ➪ [1, 10]

    // --- Intervall von 2 inklusive der Grenze 2
    out.println(ns.tailSet("2"));   ➪ [2, AB, ABC, Abc, a, ab, z, ä]

    // --- Anhängen des Zeichens 0, somit exklusive 2 (wie HeadSet)
    out.println(ns.tailSet("2\0")); ➪ [AB, ABC, Abc, a, ab, z, ä]

    // --- Intervall von..bis exklusive obere Grenze
    out.println(ns.subSet("10","Abc"));   ➪ [10, 2, AB, ABC]

    // --- Intervall von..bis inklusive obere Grenze durch Anhängen von 0
    out.println(ns.subSet("10","Abc\0")); ➪ [10, 2, AB, ABC, Abc]

    // --- holt und löscht erstes Element
    out.println(ns.pollFirst());   ➪ 1

    // --- holt und löscht letztes Element
    out.println(ns.pollLast());    ➪ ä
    out.println(ns);               ➪ [10, 2, AB, ABC, Abc, a, ab, z]

    // --- absteigender Iterator
    for (Iterator<String> i= ns.descendingIterator();
         i.hasNext();
         out.print(i.next()+" ")); ➪ z ab a Abc ABC AB 2 10
  }
}
```

Eine Diskussion der Ausgabe ist wohl nicht notwendig, da die Ausgaben die Wirkungen der Methoden recht gut zeigen.

NavigableMap

Maps eignen sich vorzüglich für In-Memory-Datenbanken. Wie bei einer RDBMS der Primärschlüssel, verbindet eine `Map` mit jedem Element einen eindeutigen Schlüssel. Ein Eintrag vom Interface-Typ `Map.Entry` ist dann ein Key-Value-Pair, verbindet also ein identifizierenden Schlüssel mit einem Wert.

Eine `Map` hat Methoden, die anstatt Elemente entweder Schlüssel oder Einträge liefert. Ansonsten sind die Methoden äquivalent zu denen von `NavigableSet`. Präfixe wie `ceiling`, `floor` oder `higher` haben bei Maps die gleiche Bedeutung wie bei Sets. Gleiches gilt für Methoden wie `pollFirstEntry`, `pollLastEntry` bzw. `headMap`, `subMap` oder `tailMap`.

Für ein Beispiel zu einer Map deklarieren wir eine kleine Klasse `Airport`, die nur das Ident und den Namen eines Flughafens kapseln soll.

```
public class Airport {

  // --- das Ident eines Flughafens besteht aus einem dreistelligen Akronym,
  //     das international festgelegt ist.
  private final String id;
  private String name;

  public Airport(String id, String name) {
    this.id=id;
    this.name= name;
  }

  public String getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name= name;
  }

  @Override
  public String toString() {
    return id+": "+name;
  }
}
```

Anhand einer `Airport`-Map werden einige Methoden der `Map` demonstriert. Der Schlüssel bzw. Key ist das Ident des Flughafens, der Wert die Airport-Instanz selbst.

Listing 1.10 NaviableMap<String,Airport>

```
package kap01;

import java.util.*;
import static java.lang.System.out;
import static java.util.Map.Entry;

public class TestNavMap {
  public static void main(String... args) {
    NavigableMap<String,Airport> aMap=
                                  new TreeMap<String,Airport>();
    // --- drei Flughäfen:
    aMap.put("AAL",new Airport("AAL","Aalborg"));
    aMap.put("TXL",new Airport("TXL","Berlin"));
    aMap.put("SHA",new Airport("SHA","Shanghai"));

    out.println(aMap);
       ➪ {AAL=AAL: Aalborg, SHA=SHA: Shanghai, TXL=TXL: Berlin}
    out.println(aMap.headMap("S") +" und "+
                aMap.tailMap("S"));
       ➪ {AAL=AAL: Aalborg} und {SHA=SHA: Shanghai, TXL=TXL: Berlin}

    // --- ohne statischen Import müsste man Map.Entry schreiben:
    Entry<String,Airport> entry= aMap.lastEntry();

    out.println(entry);                    ➪ TXL=TXL: Berlin
    out.println(aMap.ceilingEntry("P"));   ➪ SHA=SHA: Shanghai
    out.println(aMap.descendingMap());
       ➪ {TXL=TXL: Berlin, SHA=SHA: Shanghai, AAL=AAL: Aalborg}

    Set<String> kSet= aMap.descendingKeySet();
    out.println(kSet);                     ➪ [TXL, SHA, AAL]

    // --- beide nachfolgenden Operationen löschen Einträge in der Map
    kSet.remove("AAL");

    out.println(aMap.pollLastEntry());
    out.println(aMap);                     ➪ {SHA=SHA: Shanghai}

    // --- Löst zur Laufzeit eine Ausnahme aus:
    entry.setValue(new Airport("AAR","Aarhus"));
                                ➪ ... UnsupportedOperationException
  }
}
```

Leitet man eine Set von Map ab, so basiert sie auf den Original-Einträgen der Map. Die Methode `remove()` wirkt sich daher auch auf die `Map` aus. Versucht man dagegen, Ändernungen über einen Entry-Eintrag vorzunehmen, wird eine `UnsupportedOperationException` ausgelöst. Dies zeigt die letzten Anweisung.

1.11.6 ConcurrentSkipListSet

Zu `NavigableSet` gibt es eine Implementatierung `ConcurrentSkipListSet`. Das Hinzufügen bzw. Entfernen einzelner Element ist atomar, d.h. logisch unteilbar. Atomare Operationen können nicht unterbrochen werden, laufen also entweder ganz oder gar nicht ab. Im Normalfall wird dies dadurch erreicht, dass die komplette Methode synchronisiert wird. Es gibt allerdings auch Alternativen (wie beispielsweise die CAS-Implementierung von `ConcurrentStack` in Abschnitt 1.11.3).

Auch bei der `ConcurrentSkipListSet` ist das Einfügen, Entfernen und Zugreifen auf Elemente aus verschiedenen Threads parallel möglich. Ein Einfügen von `null` ist wie bei (nahezu) allen Concurrent-Implementierungen nicht möglich, da `null` als Rückgabewert signalisiert, dass es keinen Wert mehr gibt.

Nicht atomar sind bei dieser Implementierung die sogenannten Bulk- bzw. Massen-Operationen wie `addAll()`, `removeAll()`, `containsAll()` und `retainAll()`. Iteriert man also in einer Thread über die Set, während man parallel in einer anderen diese Methoden aufruft, ist das Ergebnis nicht *deterministisch*. Deterministisch nennt man Operationen, die vorhersehbar sind und durch wiederholte Aufruf reproduziert werden können. Der Vorteil von nicht deterministischen Methoden liegt insbesondere darin, dass sie keine Ausnahmen auslösen. Das gilt allerdings nicht bei dem Versuch, eine `null` als Wert einzufügen.

Die zugehörigen Iterationen nennt man *weakly consistent*. Der Begriff „schwach konsistent" besagt, dass Iterationen neben den anderen Operationen parallel ablaufen und die Elemente liefert, die der Iterator nach seiner Erschaffung in der Set vorfindet. Elemente, die vom Iterator geliefert werden, können durchaus zwischenzeitlich gelöscht worden sein. Andererseits können bereits neue Elemente in der Set eingefügt worden sein, die der Iterator nicht unbedingt anzeigt.

Das folgende Listing zeigt die Parallelverarbeitung anhand von drei Threads, die nicht fair gestartet werden. Der Ablauf ist sehr stark vom Scheduler abhängig und dieser wiederum vom Betriebssystem. Deshalb wurde noch eine zufällige Schlafzeit zwischen `0` und `TIME` in jedes `run()` eingebaut.[21]

[21] Mit 5 msec ist TIME nicht unbedingt optimal gewählt, da u.a. bei MS Windows XP die Latency (genauer wohl der periodic clock interrupt) normalerweise etwa 10 msec betragen soll.

Listing 1.11 Parallelverarbeitung in Concurrent Set

```
public class TestConNavSet {
  static Random random= new Random();
  static final int TIME= 5;

  static void sleep() {
    try { Thread.sleep(random.nextInt(TIME));
    } catch (Exception e) {}
  }

  public static void main(String... args) {
    final NavigableSet<String> cSet=
                    new ConcurrentSkipListSet<String>();
    // --- erste Thread: bulk-Operation, Einfügen von 7 Elementen
    new Thread (new Runnable() {
      public void run() {
        sleep();
        cSet.addAll(Arrays.asList("4","7","3","1","5","2","6"));
      }
    }).start();

    // --- zweite Thread: Iterieren über die gleiche Menge
    new Thread (new Runnable() {
      public void run() {
        sleep();
        // --- da weakly consistent, ist die Iteration/Ausgabe nicht vorhersehbar
        for (Iterator<String> i=
             cSet.descendingIterator(); i.hasNext();)
          System.out.print(i.next()+" "); ➪ 3 2 1
                                          ➪ 7 6 5 4 3 2 1
                                          ➪ 7 6 3 2 1
        /* auch leere Ausgabe möglich */  ➪
      }
    }).start();

    // --- dritte Thread: bis auf "1","2","3" alle Elemente löschen
    new Thread (new Runnable() {
      public void run() {
        sleep();
        cSet.retainAll(Arrays.asList("1","2","3"));
      }
    }).start();
  }
}
```

Obwohl nach Ablauf der drei Threads die Menge `cSet` nur die `String`-Elemente `"1"`,`"2"` und `"3"` enthält, sind Ausgaben zwischen den Extremen „leer“ und der Anzeige der gesamten Menge möglich.

1.11.7 ConcurrentMap

Eine `ConcurrentMap` ist erste Wahl, sofern Maps parallel bearbeitet werden müssen. Synchronisierte Wrapper einer normalen Map sind bei parallelem Zugriff aus mehreren Threads in der Performanz unterlegen. Es gibt zwei Implementierungen: `ConcurrentHashMap` und `ConcurrentSkipListMap`. Abgesehen von den Besonderheiten einer Map entspricht das Verhalten von `ConcurrentSkipListMap` der oben besprochenen `ConcurrentSkipListSet`.

ConcurrentNavigableMap

Das Interface überschreibt an sich nur covariant[22] die bereits vorhandenen Methoden der Super-Interfaces wie `SortedMap` oder `NavigableMap`. Liefert beispielsweise die Methode

```
SortedMap<K,E> headMap(K toKey)
```

so wird sie überschrieben mit

```
ConcurrentNavigableMap<K,E> headMap(K toKey)
```

Die einzige Implementierung zu dieser `Map` ist die Klasse `ConcurrentSkipListMap`. Alle Ausführungen, die zu `ConcurrentSkipListSet` im letzten Abschnitt gemacht wurden, lassen sich auf `ConcurrentSkipListMap` übertragen. Deshalb ist ein weiteres Beispiel wohl nicht notwendig.

ConcurrentHashMap

Abschließend darf ein Verweis auf die Implementierung `ConcurrentHashMap` des Interfaces `ConcurrentMap` nicht fehlen, auch wenn es sie bereits seit Java 5 gibt. Alle Operationen sind nicht-blockierend. Per Default können 16 Threads die Map parallel bearbeiten (löschen/ändern/einfügen), ohne sich also gegenseitig wie bei der Synchronisation zu blockieren. Der parallele Zugriff auf Einträge aus anderen Threads beruht auf der letzten abgeschlossenen Bearbeitung. Für Bulk-Operationen und den (zwangsläufig) schwach konstistenten Iterationen gelten die Aussagen aus Abschnitt 1.11.6. Die Anpassungen an den speziellen Einsatz, d.h. auch die parallele Arbeitslast lässt sich über die Konstruktoren steuern. Der folgende Konstruktor lässt eine ausreichende Adjustage zu:

```
ConcurrentHashMap (int initialCapacity,
                   float loadfactor,
                   int concurrencyLevel);
```

[22] Der Rückgabe-Typ einer Methode wird beim Überschreiben der Methode in einen Subtyp abgeändert. Covarianz wird ausführlich im 2. Kapitel besprochen.

Die `initialCapacity` ist dann optimal, wenn sie der benötigten Aufnahmekapazität der `Map` entspricht. Der `loadfactor` gibt an, wann eine Anpassung der Größe der `Map` erfolgen soll. Dabei bedeutet ein `loadfactor` von `0.9` dass eine Überschreitung des Füllungsgrads von 90% zu einer Größenanpassung der Map führt. Der `concurrencyLevel` ist dann optimal, wenn sie die Maximalanzahl der Threads trifft, die parallel die Map bearbeiten. Die nur lesenden Threads zählen dabei nicht mit. Einfach einen zu großen Wert zu nehmen, ist nicht vorteilhaft. Wenn wir nur ein oder zwei Threads in der Map löschen oder einfügen, sollte auch der `concurrencyLevel` entsprechend auf 1 oder 2 gesetzt werden. Das erhöht mit Sicherheit die Performanz. Der No-Args-Konstruktor

```
ConcurrentHashMap ();
```

hat folgende Defaults: `initialCapacity` von `16` Einträgen (entries), `loadFactor` von `0.75` und einen `concurrencyLevel` von `16`. Er ist ein feines kleines Beispiel für *Convention over Configuration*, ein Begriff, der mit Ruby On Rails en vogue wurde.

1.11.8 CopyOnWrite

Eine Eigenschaft wie die schwache Konsistenz der Iteratoren kann in bestimmten Fällen ungemein stören. Vor allem dann, wenn es – wie bei Observern, Listenern oder Handlern – notwendig ist, über eine temporär fixe Menge bzw. Liste iterieren zu müssen. Dann dürfen parallel laufende Bearbeitungen diesen Ablauf nicht stören. Genau für diesen Fall gibt es seit Java 5 zwei spezielle Klassen. Da die eine Klasse `CopyOnWriteArraySet` intern auf der `CopyOnWriteArrayList` basiert, genügt es, die Listen-Version zu betrachten.

CopyOnWriteArrayList

Wie der Name schon sagt, ist diese Implementierung eine besondere Concurrent-Variante von `ArrayList`. Alle Operationen, die die Liste verändern, führen intern zu einem neuen Array. Sofern aber die Liste nicht groß wird und ein stabiles Iterieren der Liste zu den Hauptaufgaben zählt, überwiegen die Vorteile gegenüber den Geschwindigkeitseinbußen. Im folgenden kleinen Beispiel werden Einfüge- und Lösch-Operationen während der Iteration durchgeführt. Die Iteration wird durch `sleep()`-Aufrufe verzögert, um die stabile Iteration zu demonstrieren.

Listing 1.12 Snapshot-Iterator bei CopyOnWriteArrayList

```
public class TestCoWList {

  static void sleep(int time) {
```

```
    try {
      Thread.sleep(time);
    } catch (Exception e) {}
  }
  public static void main(String... args) {
    final List<String> cowLst=
      new CopyOnWriteArrayList<String>(
                                    Arrays.asList("1","2","3"));

    // --- Ausgabe der drei Elemente mit anschliessend 100 ms Verzögerung
    new Thread (new Runnable() {
      public void run() {
        System.out.println("Start Iterator");
        for (String s: cowLst) {
          System.out.println(s);
          sleep(100);
        }
      }
    }).start();

    // --- nach 50 ms: Löschen der  "3"  und Einfügen von "4"..."8"
    sleep(50);
    cowLst.remove(2);
    for (char c='4'; c<'9'; c++)
      cowLst.add(Character.toString(c));
    System.out.println(cowLst);
    System.out.println("Ende main");
  }
}
```

Ausgabe zu `TestCoWList`:

```
Start Iterator
1
[1, 2, 4, 5, 6, 7, 8]
Ende main
2
3
```

Die Ausgabe zeigt den stabilen Snapshot, eine Momentaufnahme, die der Iterator vornimmt. Die Iteration findet während der Veränderung der Liste statt und ist deterministisch.

1.1 Fazit

Die einfachen Spracherweiterungen in Java 5 sind wirklich nicht spektakulär, aber durchaus angenehm. Der Code wird einfacher und teilweise auch eleganter. Mit Enumerationen sind erstmalig typ-sichere Aufzählungen möglich, die vorher mühsam als `final static` Felder deklariert werden mußten. Wie wirkungsvoll die `enum`-Implementierung gegenüber der von C++ bzw. C# wirklich ist, wird erst klar, wenn sie im Rahmen von Generics noch einmal im Detail behandelt wird. Java 6 besticht durch Detailänderungen. Insbesondere sind hier die Ergänzungen zu Floating-Point sowie die API-Erweiterungen im Concurrent-Package sehr gut gelungen. Selbst die kleinen Ergänzungen bei den Arrays sind für die tägliche Arbeit hilfreich. Einziger Kritikpunkt ist das kuriosen Console-I/O. Es ist ziemlich unverständlich, dass eine Bibliothek wie in C in der Standard Edition nicht realisiert werden kann.

1.12 Referenzen

- Gosling, J., Joy, B., Steele, G., Bracha, G. (2005). *The Java Language Specification*. 3rd Ed., Prentice Hall PTR
- Bloch, Joshua (2001). *Effective Java Programming Language*. Addison-Wesley, Amsterdam
- Goetz, Brian (2006). *Java Concurrency in Practice*. Addison-Wesley Professional
- Kurniawan,Budi (2007). *Java 6 New Features: A Tutorial*. Brainysoftware.com, Canada
- JSR 270: *JavaTM SE 6 Release Contents*. `http://jcp.org/en/jsr/detail?id=270`
- JSR 166: *Concurrency Utilities* `http://jcp.org/en/jsr/detail?id=166`
- Sun Online Documentation: *Java Programming Language*. `http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html`
- Cowlishaw, M., Bloch, J., Darcy, J.D. (2004). *Fixed, Floating, and Exact Computation with Java's Bigdecimal*. Dr. Dobb's Journal/Portal. `http://www.ddj.com/java/184405721`
- Leavens, Gery T. (2006). *Not a Number of Floating Point Problems*. Journal of Object Technology, Vol. 5, No. 2