Programmiersprache

Java

2021 / Teil 4

Detlef Wilkening
www.wilkening-online.de
© 2021

Programmiersprache Java

8 Funktionen 2

8.1 Funktions-Arten...3
8.2 Syntaktischer Aufbau 3
8.3 Variablen in Funktionen..4
8.4 Funktions-Parameter...5
8.5 Funktions-Rückgaben 6
8.6 Überladen 7
8.7 Rekursion..8

9 Ausgewählte Bibliotheks-Klassen...9

9.1 Klasse String..9
9.2 Klassen StringBuilder & StringBuffer..13
9.3 Container & Iteratoren...13
9.4 Stream-API und Lambdas 27
9.5 Wrapper Klassen 31
9.6 Zufalls-Zahlen 33
9.7 Datum und Uhrzeit 34
9.8 Datei- und Verzeichnis-Handling 35

8 Funktionen

In Java gibt es keine freien (globalen) Funktionen – alle Funktionen sind immer einer Klasse zugeordnet. Dies gilt auch für die Main-Funktionen.

Syntax:

```
[Modifizierer] <Rückgabetyp> <Fkt-Name> ( [Parameterliste] ) {
   <Implementierung>
}
```

- Modifizierer – z.B.: public, private, static, final, ...
- Eine Parameterliste ist eine durch Komma getrennte Auflistung von Parametern (sie kann auch leer sein. Ein Parameter besteht aus Typ und Name.

Bsp:

```
public static void f() { ... }
protected final int doit(StringBuffer sb) { ... }
private static String calcName(String s, int i) { ... }
```

Aufruf:

```
<Fkt-Name>( <Argumentliste> );
```

Bsp:

```
f();
calcName("", 1);
```

Rückgaben können ignoriert werden – siehe Bsp „calcName“.

Bemerkung – fast jeder ausführbare Code steht immer in einer Funktion. Die einzigen Ausnahmen sind Initialisierungs-Blöcke, die aber im Prinzip auch nur sehr spezielle Funktionen sind.

Bemerkung – Funktions-Namen werden in Java per Konvention klein begonnen, und dann kapitalisiert fortgesetzt – vergleiche z.B. den Namen „calcName“.

8.1 Funktions-Arten

Achtung – in Java gibt es zwei Arten von Funktionen:

- Element-Funktionen
- Klassen-Funktionen

Klassen-Funktionen sind Funktionen, die mit dem Modifizierer „static“ definiert sind – wie z.B. die Funktion „main“ in unseren Beispielen. Nur Klassen-Funktionen können direkt aufgerufen werden. Element-Funktion benötigen einen Objekt-Bezug, und werden dann mit dem Objekt und dem Punkt-Operator „.“ aufgerufen.

Für Objekte z.B. vom Typ „String“ benutzen wir die Element-Funktionen einfach. Im Detail kennen lernen werden wir sie erst mit der Einführung von Klassen und Objekten in Kapitel todo. Solange sollten Sie in ihren Beispielen nur Klassen-Funktionen definieren, d.h. jede selbstgeschriebene Funktion sollte den Modifizierer „static“ bekommen.

```
public class Appl {

   public static void sf() { ... }

   public void f() { ... }

   public static void main(String[] args) {
      sf();                                    // Aufruf geht, da sf static ist
      f();                                     // Compiler-Fehler, da Element-Fkt

      String s = "Hallo";
      s.length();                              // Okay - Aufruf von Element-Fkt
   }                                           //        mit Objekt-Bezug

}
```

8.2 Syntaktischer Aufbau

Eine Java Funktion besteht aus einer Folge von Anweisungen. Eine Anweisung ist:

- eine Instruktion (wird durch Semikolon beendet – z.B. Zuweisung, Definition., Funktionsaufruf,...),

- eine Kontrollstruktur (if, while, switch,...), oder
- ein Block.

Ein Block ist eine Folge von Anweisungen, die durch die geschweiften Klammern eingeschlossen sind – ein Block ist immer auch ein Scope, d.h. ein Sichtbarkeitsbereich von Variablen (siehe Kapitel 8.3).

8.3 Variablen in Funktionen

Variablen können an (fast) jeder beliebigen Stelle in Funktionen definiert werden. Dabei werden sogenannte lokale Variablen erzeugt, die lokal zur Funktion sind.

```
void fct() {
   int var = 23;
   System.out.println(var);
}
```

Ein einmal vergebener Name darf im gleichen Block nicht ein zweites Mal benutzt werden – ansonsten wäre der Name ja nicht eindeutig.

```
void fct() {
   int var = 23;
   System.out.println(var);
   int var = 42;                // Compiler-Fehler - var schon definiert
}
```

Wird eine Variable innerhalb eines Blocks in einer Funktion definiert, so ist sie auch nur innerhalb dieses Blocks sichtbar. Ein Block bildet also einen Sichtbarkeitsbereich.

```
void fct() {
   while (true) {
      int var = 87;
      System.out.println(var);   // Okay - var ist sichtbar
      break;
   }
   System.out.println(var);      // Compiler-Fehler - var hier nicht sichtbar
}
```

Eine Besonderheit sind hierbei Variablen-Definitionen im Initialisierungs-Teil einer For-Schleife. Obwohl sie nicht in einem speziellen Block definiert sind, sind diese Variablen nur in der Schleife bekannt.

Ist eine lokale Variabel sichtbar (d.h. sie befinden sich innerhalb des definierenden Scopes in der Funktion), so darf auch in verschachtelten Blöcken keine weitere Variable des gleichen Names definiert werden.

```
int n = 4;
{
   int n = 3;            // Compiler-Fehler - Variable n ist schon definiert.
}
```

8.3.1 Modifier „final“ für lokale Variablen

Der einzig erlaubte Modifizierer für Variablen in Funktionen ist der Modifizierer „final“. Wird er

angegeben, so kann die Variable nicht verändert werden.

```
final int n = 3;
n = 4;                 // Compiler-Fehler - Variable darf nicht geaendert werden

final StringBuffer sb = new StringBuffer("StringBuffer");
sb = new StringBuffer("Error");          // Compiler-Fehler - Variable ist final

sb.append(" das geht aber");             // Okay, final schuetzt nur die Variable
                                         //      nicht das referenzierte Objekt
```

Achtung – bei Referenz-Variablen bezieht sich „final“ nur auf die Referenz, nicht auf das referenzierte Objekt – siehe auch das vorherige Beispiel.

8.4 Funktions-Parameter

Funktions-Parameter sind eigentlich ganz normale lokale Variablen, die beim Funktions-Aufruf mit den Argumenten des Aufrufers initialisiert werden. Damit unterliegen sie natürlich auch der Wert- bzw. der Referenz-Semantik, und das ist für Funktions-Parameter interessant.

Alle elementaren Datentypen werden per Wert-Semantik, d.h. mit einer echten Kopie, übergeben. Änderungen innerhalb der Funktion haben nur Einfluss auf die Kopie in der Funktion, und betreffen das Original beim Aufrufer nicht.

```
void f1(int n1) {
   System.out.println("=> f1");
   System.out.println("   " + n1);
   n1 += 12;
   System.out.println("   " + n1);
   System.out.println("<= f1");
}

int n = 23;
System.out.println(n);
f1(n);
System.out.println(n);
```

```
Ausgabe
23
=> f1
   23
   35
<= f1
23
```

Anders ist dies dagegen bei Referenz-Variablen, d.h. bei allen Variablen, deren Typen keine elementaren Datentypen sind. In diesem Fall sind auch Funktions-Parameter Referenz-Variablen, und sie enthalten nur eine Referenz auf das eigentliche Objekt. Eine Funktion kann also das referenzierte Objekt des Aufrufers verändern!

```
void f1(StringBuffer sb1) {
   System.out.println("=> f1");
   System.out.println("   " + sb1);
   sb1.append(" Welt");
   System.out.println("   " + sb1);
   System.out.println("<= f1");
}

StringBuffer sb = new StringBuffer("Hallo");
```

```
System.out.println(sb);
f1(sb);
System.out.println(sb);
```

```
Ausgabe
Hallo
=> f1
   Hallo
   Hallo Welt
<= f1
Hallo Welt
```

Achtung – Sie können dieses Verhalten nicht beeinflussen. Die Übergabe-Semantik des Funktions-Parameters liegt automatisch mit dem Typ des Parameters fest.

8.5 Funktions-Rückgaben

Hat eine Funktion den Rückgabe-Typ „void“, so gibt sie nichts zurück.

```
void f() {
}
```

Soll sie dagegen etwas zurückgeben, so muss der entsprechende Rückgabe-Typ in der Funktions-Definition angeben werden. Hat eine Funktion einen von „void“ verschiedenen Rückgabe-Typ, so muss sie mit einer Return-Anweisung beendet werden, die den Rückgabe-Wert definiert. Der Rückgabe-Wert in der Return-Anweisung muss natürlich zum Rückgabe-Typ der Funktion passen.

```
int f1() {
   return 12;
}

String f2() {
   return "Java";
}

StringBuffer f3() {
   StringBuffer sb = new StringBuffer("Text");
   return sb;
}

List<String> f4() {
   return null;
}
```

Eine Funktion kann beliebig viele Ausgänge haben. Jede Return-Anweisung beendet die Funktion direkt, und gibt den ausgewerteten Ausdruck der Return-Anweisung zurück.

```
int sgn(int n) {
   if (n<0) {
      return -1;
   }
   return n==0 ? 0 : 1;
}

System.out.println("-6 => " + sgn(-6));
System.out.println("-1 => " + sgn(-1));
System.out.println(" 0 => " + sgn( 0));
System.out.println(" 1 => " + sgn( 1));
System.out.println(" 3 => " + sgn( 3));
```

```
Ausgabe
-6 => -1
-1 => -1
 0 => 0
 1 => 1
 3 => 1
```

Eine Void-Funktion, d.h. eine Funktion ohne Rückgabe, kann jederzeit mit einer leeren Return-Anweisung beendet werden.

```
void f(int i) {
   if (i<12) {
     return;                    // Leere Return-Anweisung
   }
   System.out.println(i);
}
```

Funktionen können jeden beliebigen Typ zurückgeben, sowohl elementare als auch benutzerdefinierte Typen. Hierbei unterliegen natürlich auch die Funktions-Rückgaben der entsprechenden Wert- bzw. Referenz-Semantik.

Funktions-Rückgaben können direkt benutzt werden, d.h. werden Objekte wie z.B. Strings zurückgegeben, so können für sie direkt Element-Funktionen aufgerufen werden. Man nennt dies auch Funktions-Verkettung.

```
String fct() {
   return "Java";
}

int len1 = fct().length();       // Aufruf der Funktion 'length()' fuer den
System.out.println(len1);        // zurueckgegebenen String

String s = fct();                // So haette man es auch machen koennen
int len2 = s.length();           // Mit lokaler Zwischen-Variable 's'
System.out.println(len2);
```

```
Ausgabe
4
4
```

8.6 Überladen

Funktionen können überladen werden, d.h. der Funktions-Name darf mehrfach benutzt werden, solange sich die Funktionen durch ihre Parameter-Typen unterscheiden, d.h. der Compiler den Aufruf eindeutig zuordnen kann. Möglich ist dabei auch die Variation der Anzahl an Parametern – auch dort ist die Funktion exakt erkennbar. Der Rückgabetyp trägt **nicht** zur Unterscheidung bei.

```
public class A {

   public static void f() {
      System.out.println("---");
   }

   public static void f(int arg) {
      System.out.println("int: " + arg);
   }

   public static void f(String arg) {
      System.out.println("String: " + arg);
```

```
   }

   public static void f(int arg1, String arg2) {
      System.out.println("int, String: " + arg1 + ", " + arg2);
   }

   public static void main(String[] args) {
      f();
      f(8);
      f("Willy");
      f("42");            // Ist natuerlich auch ein String, kein 'int'
      f(3, "Java");
   }

}
```

```
Ausgabe
---
int: 8
String: Willy
String: 42
int, String: 3, Java
```

Hinweis – das Feature „Überladen“ funktioniert mit allen Arten von Funktionen, also sowohl mit Klassen-Funktionen, Element-Funktionen, als auch mit Konstruktoren.

8.7 Rekursion

Wenn eine Funktion im gleichen Thread mehrfach **ineinander** (d.h. nicht nacheinander, sondern gleichzeitig parallel) aufgerufen wird, nennt man das „Rekursion“ bzw. „rekursive Programmierung“.

Eine Funktion muss sich dabei nicht zwangsläufig selber aufrufen, sondern dies kann auch über mehrere Zwischenstationen passieren, z.B. „f() => g() => h() => f()“. Solche Fälle sind häufig gar nicht mehr sofort zu erkennen, von daher passiert dies in der Praxis häufiger, als man vielleicht im ersten Augenblick denkt.

Es gibt aber auch viele Probleme, die sich rekursiv viel viel leichter programmieren lassen als ohne Rekursion. Hier ein Beispiel, das man in der Praxis sicher nicht rekursiv lösen würde – die Summe aller Zahlen von 1 bis n.

```
public static long sum(long arg) {
   if (arg<=1) {              // Operator <= statt == um die Funktion bei
fehlerhaften
      return 1;               // Argumenten (arg<1) sauber zu beenden.
   }
   long erg = arg + sum(arg-1);
   return erg;
}
```

Hierbei wird quasi direkt die mathematische Definition umgesetzt:

$sum(1) := 1$

$sum(n) := n + sum(n-1)$

Natürlich lässt sich die Summe der Zahlen von 1 bis n direkt über die Gaußsche-Summen-Formel „n*(n+1)/2“ berechnen – was hier auch viel sinnvoller wäre – aber es geht eben auch

rekursiv.

Hinweis – es lässt sich übrigens beweisen, dass jedes Problem was iterativ gelöst werden kann (d.h. mit einer Schleife) sich auch rekursiv lösen lässt, und umgekehrt.

Praxis – in den aller-meisten Fällen sind die iterativen Lösungen schneller und benötigen weniger Speicher – sie sind daher vorzuziehen. Ein Schleifen-Durchlauf ist schnell und effizient, während ein Funktions-Aufruf doch relativ *teuer* ist (bezogen auf die Performance).

In so manchen Fällen ist die rekursive Lösung aber ein 5-Zeiler, während die iterative Lösung Tage harter Arbeit sein kann und hinterher aus vielen Quelltext-Zeilen besteht – Beispiele für Lösungen, die rekursive erstaunlich einfach sind, finden sich z.B. bei der rekursiven Datei-Suche, oder in der Musterlösung Aufgabe „Türme von Hanoi“.

9 Ausgewählte Bibliotheks-Klassen

Die Java Bibliothek umfasst über 4500 Klassen. Für viele Probleme stehen daher fertige Klassen zur Verfügung. Ein paar ganz allgemeine und sehr zentrale Klassen möchte ich hier ganz kurz vorstellen, damit wir sie in späteren Beispielen bzw. dem Praktikum benutzen können. Aber machen Sie sich klar, es sind nur ganz wenige aus dem großen Angebot der Java-Welt.

9.1 Klasse String

Mit der Klasse String werden konstante Unicode Zeichenketten repräsentiert. Ja, Sie haben richtig gelesen: „konstante Zeichenketten“. Strings können aus Performance-Gründen nicht verändert werden. Wollen Sie Zeichenketten modifizieren, so müssen Sie die Klassen StringBuilder oder StringBuffer verwenden.

Hinweise:

- Man nennt Strings, wie auch andere unveränderbare Objekte, auch „immutable“ Objekte.
- Strings können einfach so benutzt werden, und benötigen kein „import“. Sie kommen aus dem Package „java.lang“, das immer automatisch zur Verfügung steht.

9.1.1 Initialisierung

Da Zeichenketten in der Praxis sehr häufig vorkommen, sind in der Sprache Java einige Unterstützungen für Strings integriert, die über eine normale ‚Klasse‘ hinausgehen.

So können Strings neben dem expliziten Erstellen mit „new“ auch einfach mit Zeichenketten erzeugt werden.

```
public class Kap_09_01_Bsp_01_Strings {

   public static void main(String[] args) {
```

```
      String s1 = new String("Java");  // So muessen in Java eigentlich alle
      System.out.println(s1);          // Objekte angelegt werden: mit "new"

      String s2 = "Hallo Welt";        // Aber bei Strings geht es auch ohne
      System.out.println(s2);          // "new" - Nettigkeit der Sprache

      s1 = "Viel einfacher so";        // Und das geht auch bei Zuweisungen
      System.out.println(s1);

      s1 = new String("Aufwaendig");   // Die normale Art fuer Nicht-Strings
      System.out.println(s1);
   }

}
```

```
Ausgabe
Java
Hallo Welt
Viel einfacher so
Aufwaendig
```

9.1.2 Operator +

Für die Klasse String ist in der Sprache Java der Operator + definiert. Auf zwei Strings angewandt ist das Ergebnis ein neuer String, der aus den beiden konkatinierten Operanden besteht.

```
public class Kap_09_01_Bsp_02_StringAddition1 {

   public static void main(String[] args) {
      String s1 = "Hallo";
      String s2 = " Welt";
      String s = s1 + s2;
      System.out.println(s);
   }

}
```

```
Ausgabe
Hallo Welt
```

Wie schon kurz in Teil 1 ist nur einer der Operanden ein String, so wird der andere implizit in einen String umgewandelt. Wir sehen dies im folgenden Quelltext:

- in der Zeile (*) für einige elementare Datentypen („boolean“, „int“ und „long“),
- in der Zeile (**) nochmal expliziter für ein „int“, und
- in der Zeile (****) für ein AWT-Label. Wir werden GUI-Label später näher kennen lernen – hier steht es einfach für ein komplexes Objekt. Als Beispiel, dass wirklich jedes Objekt beim Operator + in einen String gewandelt wird.

```
import java.awt.Label;

public class Kap_09_01_Bsp_03_StringAddition2 {

   public static void main(String[] args) {
      String s = "Variablen: ";
      boolean b = true;
      int i = 12345;
      long l = 123456789012345L;
      s = s + b + "=" + false + " - i=" + i + " - l=" + l;  // Zeile (*)
      System.out.println(s);

      int n = 42;
      String s1 = "" + n;                                    // Zeile (**)
```

```
      String s2 = "" + Integer.toString(n);              // Zeile (***)
      System.out.println(s1);
      System.out.println(s2);

      Label label = new Label("Beschriftung");
      s1 = "" + label;                                   // Zeile (****)
      s2 = label.toString();                             // Zeile (*****)
      System.out.println(s1);
      System.out.println(s2);
   }

}
```

```
Ausgabe
Variablen: true=false - i=12345 - l=123456789012345
42
42
java.awt.Label[label0,0,0,0x0,invalid,align=left,text=Beschriftung]
java.awt.Label[label0,0,0,0x0,invalid,align=left,text=Beschriftung]
```

Alternativ zu den impliziten Wandlungen beim Operator + in z.B. Zeile (**) und (****) sind auch die mehr expliziten Wandlungen wie hier in den Zeilen (***) und (*****) möglich.

- Für alle elementaren Datentypen existieren Standard-Umwandlungen, die sich auch in den jeweiligen Wrapper-Klassen wiederfinden – siehe später im Kapitel.
- Für alle anderen Typen kann die Elementfunktion „toString()“ benutzt werden, die für alle Klassen definiert ist.

Bitte beachten Sie: auch wenn im Quelltext des letzten Beispiels „s=...“ steht, so wird nicht der String verändert. Es ändert sich nur der Wert der Referenz-Variablen „s“, die nach der Zuweisung auf ein anderes String-Objekt verweist. Das originale String-Objekt mit dem Wert „Variablen: “ wird nicht verändert, sondern jetzt nur nicht mehr referenziert.

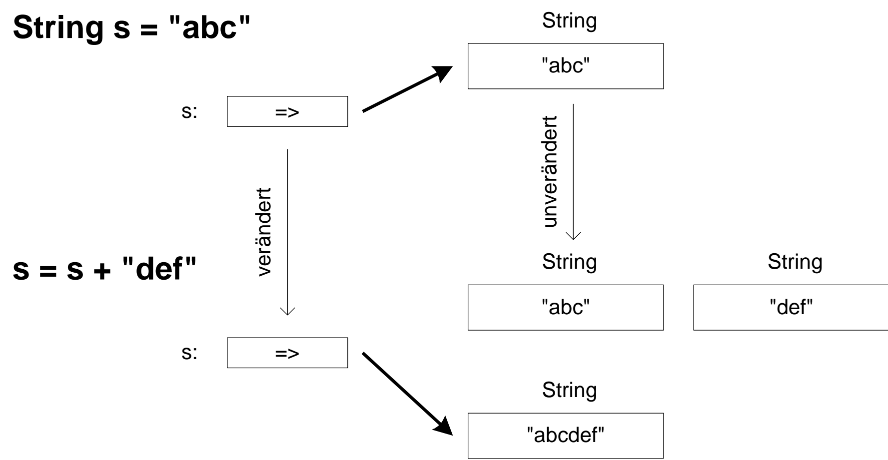

Abb. 9-1 : Ein String-Objekt wird nicht verändert

Hinweis – die Addition von Strings mit dem Plus-Operator ist nicht sehr effizient. Wie Sie an der obigen Abbildung sehen können, müssen bei dieser Operation neue Objekte angelegt und die Zeichen kopiert werden – was relativ viel Zeit kostet. Nutzen Sie aufgrund der Performance Vorteile bei String-Verkettungen die Klassen „StringBuilder“ und „StringBuffer“.

9.1.3 Referenz-Semantik und Konstante Zeichenketten

Da String eine Klasse und kein elementarer Datentyp ist, unterliegen String-Variablen natürlich der Referenz-Semantik.

Und die Klasse String repräsentiert konstante Zeichenketten. D.h. es gibt keine Möglichkeit einen String zu verändern. Alle Operationen auf Strings lassen diese unangetastet, bzw. geben einen neuen zurück – der Originalstring bleibt immer unverändert, String-Objekte sind immutable. Damit entfällt das typische *Problem* der Referenzsemantik, nämlich dass ein Objekt von woanders einfach geändert wird. Strings sind dagegen resistent, und darum an vielen Stellen unproblematischer.

Achtung – umgekehrt gibt es bei Strings dadurch auch eine Falle: der Vergleichs-Operator „==“ führt auch bei Strings nur einen Identitäts-Vergleich durch, und keinen Wert-Vergleich. Leider verhält sich der Operator „==“ bei Strings aber manchmal so, als würde er einen Wert-Vergleich durchführen. Vergleichen Sie Strings immer mit „equals“, auch wenn der Operator „==“ scheinbar funktioniert.

9.1.4 Schnittstelle

Die Klasse String ist in java.lang definiert. Sie hat keine public Attribute, aber viele Element-Funktionen – hier eine kleine Auswahl:

String()	Konstruktor: erzeugt einen leeren String.
String(char[ ])	Konstruktor: erzeugt einen String aus einem char-Array.
String(StringBuffer)	Konstruktor: erzeugt einen String aus einem String-Buffer.
boolean equals(String)	Gibt zurück, ob die beiden Strings wert-gleich sind.
boolean equalsIgnoreCase (String anotherString)	Vergleicht zwei String ohne Berücksichtigung von Groß- und Klein-Schreibung.
char charAt(int)	Gibt das Zeichen an der spezifizierten Position zurück.
int length()	Gibt die Anzahl an Zeichen des Strings zurück, d.h. seine Länge.
String trim()	Gibt einen neuen String zurück, der dem alten ohne führende und folgende Whitespaces (nicht sichtbare Zeichen wie z.B. Leerzeichen oder Tabulator) entspricht.
boolean startsWith(String)	Gibt zurück, ob der String mit dem übergebenen beginnt.
boolean endsWith(String)	Gibt zurück, ob der String mit dem übergebenen aufhört.
String substring (int beginIndex, int endIndex)	Gibt eine Teil-String als neuen String zurück. Der Teil-String beginnt beim Zeichen mit dem Index „beginIndex“ **inkl**. und endet beim Zeichen mit dem Index „endIndex“ **exkl**. Bei fehlerhaften Indices wird eine Exception geworfen.

Die komplette Schnittstelle der Klasse „String“ finden Sie in der offiziellen Java Referenz-

Dokumentation.

9.2 Klassen StringBuilder & StringBuffer

Die Klassen StringBuilder und StringBuffer repräsentieren Unicode Zeichenketten, die verändert werden dürfen. Beide Klassen enthalten viele Funktionen zur Modifikation von Texten, z.B. die Element-Funktion „append“, die einen Text an das bestehende StringBuilder- oder StringBuffer-Objekt anfügen.

```
public class Kap_09_02_Bsp_01_StringBuilderUndBuffer {

   public static void main(String[] args) {
      StringBuilder sb1 = new StringBuilder("1: Append");
      sb1.append(" mit");
      sb1.append(" StringBuilder");
      System.out.println(sb1);

      StringBuffer sb2 = new StringBuffer("2: Append");
      sb2.append(" mit");
      sb2.append(" StringBuffer");
      System.out.println(sb2);
   }

}
```

```
Ausgabe
1: Append mit StringBuilder
2: Append mit StringBuffer
```

Im Gegensatz zur String-Addition wird hierbei wirklich das Objekt verändert und kein Neues erzeugt. Dafür müssen StringBuilder- und StringBuffer-Objekte ganz normal explizit mit „new“ erzeugt werden.

Die Klasse „StringBuffer“ existiert seit dem JDK 1.0, während die Klasse „StringBuilder“ erst mit dem JDK 1.5 eingeführt. Beide Klassen haben dieselbe Schnittstelle. Im Gegensatz zu StringBuffer ist StringBuilder aber nicht multi-threading fest, dafür aber wesentlich schneller. Sie sollten also in single-threaded Anwendungen, oder in MT-unkritischen Situation (wie z.B. String-Manipulation innerhalb einer Funktion mit einer lokalen Variablen) die Klasse „StringBuilder“ gegenüber „StringBuffer“ bevorzugen. Benötigen Sie dagegen eine MT feste Klasse, so führt kein Weg an „StringBuffer“ vorbei – auch wenn dies mit Performance-Einbußen verbunden ist.

Hinweis – auch die Klassen StringBuilder und StringBuffer können einfach so benutzt werden, und benötigen kein „import“. Vergleichbar zur Klasse String kommen sie aus dem Package „java.lang“, das immer automatisch zur Verfügung steht.

9.3 Container & Iteratoren

Container sind Objekte, die andere Objekte aufnehmen können, diese verwalten, und dabei automatisch passend wachsen. Es gibt nicht den einen Container, der für alle Zwecke optimal geeignet ist, sondern jeder Container hat seine Vor- und Nachteile. Je nach Anwendung bzw.

Anforderung ist mal der Eine, mal der Andere besser geeignet. Nähere Informationen hierzu finden Sie in Kapitel todo, und ausführlicher in vielen Büchern über „Algorithmen und Datenstrukturen".

Seit der ersten Java Version (JDK 1.0) sind einige grundlegende Container Bestandteil der Java Klassen-Bibliothek. Mit der Version Java 2 (JDK 1.2) wurden die Container stark überarbeitet und erweitert, und dann mit jeder JDK Version weiter verbessert. So ist heute ein sehr ordentliches Container-Framework Bestandteil der Java Bibliothek.

9.3.1 Einführung in Container & Iteratoren

Ein typischer Container ist die „ArrayList". Sie wird – wie jedes Objekt in Java – mit „new" erzeugt und kann dann einfach genutzt werden. In spitzen Klammern geben wir den Typ der Elemente an, die in der ArrayList gespeichert werden sollen. Mit diesen spitzen Klammern nutzen wir die seit JDK 1.5 vorhandenen Generics, um den Container typsicher zu machen.

Im folgenden Beispiel wird eine ArrayList für Strings angelegt und mit 3 Strings gefüllt. Die aktuelle Anzahl an Elementen im Container kann man dann mit „size()" abfragen:

```
import java.util.ArrayList;

public class Kap_09_03_Bsp_01_ContainerEinstieg {

   public static void main(String[] args) {
      ArrayList<String> al = new ArrayList<String>();
      al.add("Hallo");
      al.add("Java");
      al.add("Kurs");

      System.out.println("Array-List enthaelt " + al.size() + " Elemente");
   }

}
```

```
Ausgabe
Array-List enthaelt 3 Elemente
```

Die Typisierung mit den spitzen Klammern bewirkt, dass keine Objekte falschen Typs in den Container eingefügt werden können.

```
ArrayList<String> l = new ArrayList<String>();
l.add("Java");                  // Okay, "Java" ist ein String
l.add(123);                     // Compiler-Fehler, kein String
l.add(new StringBuilder());     // Compiler-Fehler, auch kein String
```

Seit JDK 1.7 kann man sich die Erzeugung eines typisierten Objektes erleichtern – man benötigt die Angabe des Element-Typs in den spitzen Klammern nicht mehr, da der Compiler diesen aus dem Variablen-Typ ermitteln kann. Das „<>" wird auch der Diamond-Operator genannt.

```
import java.util.ArrayList;

public class Kap_09_03_Bsp_02_ContainerEinstiegJdk17 {

   public static void main(String[] args) {
      ArrayList<String> al = new ArrayList<>();
      al.add("JDK 1.7");
```

```
      System.out.println("Array-List enthaelt " + al.size() + " Element");

      al = new ArrayList<>();
      System.out.println("Array-List enthaelt " + al.size() + " Elemente");
   }

}
```

```
Ausgabe
Array-List enthaelt 1 Elemente
Array-List enthaelt 0 Elemente
```

Aber wie läuft man jetzt über einen Container und gibt z.B. alle Element im Container aus? Für eine ArrayList könnte man dies noch mit einer normalen For-Schleife mit Index-Zähler und dem wahlfreiem Zugriff auf die Array-List (Element-Funktion „get(index)") umsetzen:

```
import java.util.ArrayList;

public class Kap_09_03_Bsp_03_ContainerSchleife {

   public static void main(String[] args) {
      ArrayList<String> al = new ArrayList<>();
      al.add("Hallo");
      al.add("Java");
      al.add("Interessierte");

      // Achtung - so laeuft man eigentlich nie ueber einen Container
      for (int i = 0; i < al.size(); i++) {
         String s = al.get(i);
         System.out.print(s + " ");
      }
   }

}
```

```
Ausgabe
Hallo Java Interessierte
```

Für das normale Laufen über einen Container sollte man nie den wahlfreien Zugriff mit „get(index)" einsetzen, denn:

- Es gibt Container, bei denen der wahlfreie Zugriff sehr langsam ist – z.B. die LinkedList, siehe Kapitel 9.3.5.
- Es gibt viele Container, die prinzip-bedingt gar keinen wahlfreien Zugriff unterstützen können, und daher auch nicht enthalten – z.B. eine HashMap, siehe Kapitel 9.3.9.

Außerdem gibt es für bessere Lösungen für dieses Problem. An die Stelle des wahlfreien Zugriffs treten dann die neue For-Schleife, Iteratoren, oder seit Java 8 Streams (Kapitel 9.4). Beginnen wir mit der neuen For-Schleife.

Hinweis – die Nutzung des wahlfreien Zugriffs sollte nur dann genutzt werden, wenn man diesen explizit durch den Algorithmus benötigt. Dann ist er natürlich sehr sinnvoll. Aber Sie sollten dann auch einen Container wählen, der den wahlfreien Zugriff performant unterstützt, wie z.B. die ArrayList – siehe Kapitel 9.3.4.

9.3.1.1 Neue For-Schleife für Container

Mit JDK 1.5 (Java 5) wurde ein neuer For-Schleifentyp eingeführt – der seit dem die normale

Variante ist. Hierbei muss nur noch eine Element-Lauf-Variable mit Typ, und nach einem Doppelpunkt der Container angegeben werden – den Rest macht der Compiler automatisch – siehe Beispiel.

```
import java.util.ArrayList;

public class Kap_09_03_Bsp_04_NeueForSchleife {

   public static void main(String[] args) {
      ArrayList<String> l = new ArrayList<>();
      l.add("Java");
      l.add("mit");
      l.add("neuer");
      l.add("JDK 1.5");
      l.add("For-Schleife");

      for (String s : l) {
         System.out.print(s + " ");
      }
      System.out.println();
   }

}
```

Ausgabe
```
Java mit neuer JDK 1.5 For-Schleife
```

Diese Schleife kann eigentlich nur eins: einfach über den Container laufen – Element für Element. Mehr nicht, aber auch nicht weniger – und das ist der normale Anwendungsfall für Container mit Schleifen, der 95 % Fall. Und den beherrscht sie einfach, schnell und zuverlässig. Und darum nehmen wir diese Schleife auch immer für diesen Use-Case.

Die einzigen zusätzlichen Möglichkeiten der Container-For-Schleife sind die Nutzung der Sprung-Anweisungen „break“ und „continue“ – die die Schleife vorzeitig verlassen oder direkt zum nächsten Element gehen können. Hier ein Beispiel mit der neuen For-Schleife mit „break“ und „continue“:

```
import java.util.ArrayList;

public class Appl {

   public static void main(String[] args) {
      ArrayList<String> l = new ArrayList<String>();
      l.add("a");
      l.add("b");
      l.add("c");
      l.add("d");
      l.add("e");
      l.add("f");

      for (String s : l) {
         System.out.print(s);
         if (s.equals("b")) {
            continue;                         // Zeile (*)
         }
         System.out.print("-");
         if (s.equals("e")) {
            break;                            // Zeile (**)
         }
      }
      System.out.println();
   }

}
```

```
Ausgabe
a-bc-d-e-
```

Nach der Ausgabe von „b “ wird kein Bindestrich ausgegeben, da das „continue“ in Zeile (*) zuschlägt, und nach Ausgabe von „e-“ wird die Schleife aufgrund von „break“ in Zeile (**) abgebrochen.

9.3.1.2 Container & Iteratoren

Ein Iterator ist die Abstraktion eines Objekts mit dem über eine Objekt-Menge gelaufen (iteriert) werden kann. Ein Iterator zeigt auf ein Objekt, kann auf das nächste gesetzt werden, und weiß, ob es noch weitere Objekte in der Menge gibt.

Jeder Java Container hat eine Element-Funktion „iterator()“, die einen Iterator über den Container zurückgibt. Mit der Element-Funktion „hasNext()“ kann am Iterator abgefragt werden, ob noch weitere Elemente vorliegen. Die Element-Funktion „next()“ geht vor das nächste Element und gibt dabei das aktuelle Element zurück:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Appl {

   public static void main(String[] args) {
      ArrayList<String> al = new ArrayList<>();
      al.add("Lasst");
      al.add("uns");
      al.add("Java");
      al.add("lernen");

      // Iterator mit While-Schleife
      Iterator<String> it1 = al.iterator();
      while (it1.hasNext()) {
         String s = it1.next();
         System.out.print(s + " ");
      }
      System.out.println();

      // Iterator mit For-Schleife
      for (Iterator<String> it2 = al.iterator(); it2.hasNext(); ) {
         String s = it2.next();
         System.out.print(s + " ");
      }
      System.out.println();
   }

}
```

```
Ausgabe
Lasst uns Java lernen
Lasst uns Java lernen
```

In der Praxis trifft man Iteratoren sowohl mit While- als auch mit For-Schleife – darum enthält das obige Beispiel beide Varianten. Sie sind gleichwertig und es ist daher reine Geschmackssache, welche Variante Sie bevorzugen.

9.3.2 Untypisierte Container

Alle bisherigen Beispiele arbeiten mit typisierten Containern, d.h. Containern, deren Element-Typ mit spitzen Klammern angegeben ist. Diese Typisierung hat zwei Vorteile:

- Sie können nur Elemente passenden Typs in den Container einfügen – Objekte mit falschem Typ erzeugen einen Compiler-Fehler (siehe Beispiel weiter oben).
- Holen Sie Element aus dem Container, so ist der Typ bekannt und spezielle Konvertierungen sind nicht notwendig.

In der Praxis begegnet uns noch sehr häufig Code mit untypisierten Containern, da dies über 9 Jahre Stand der Java Technik war:

- Viele Schnittstellen sind in dieser Ära entstanden, setzen daher auf untypisierten Container auf, und sind noch heute aktuell.
- Auch heute noch viel alter Java Code exisitiert und genutzt wird.

Von daher sollten Sie auch die Verwendung von untypisierten Containern kennen – in neuem Code sollten Sie aber nur typisierte Container verwenden. Das folgende Beispiel zeigt wieder die schon bekannte Nutzung der ArrayList, diesmal aber ohne Typisierung:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Kap_09_03_Bsp_07_UntypisierteContainer {

   public static void main(String[] args) {
      ArrayList al = new ArrayList();                      // (*)
      al.add("Untypisierter");
      al.add("Container");
      al.add("-");
      al.add("ohne");
      al.add("Generics");

      Iterator it = al.iterator();                         // (**)
      while (it.hasNext()) {
         String s = (String) it.next();                    // (***)
         System.out.print(s + " ");
      }
      System.out.println();
   }

}
```

Ausgabe
```
Untypisierter Container - ohne Generics
```

Bevor wir den obigen Code besprechen, erstmal ein wichtiger Hinweis: der obige Code erzeugt sowohl im Java-Compiler „javac“ auf der Kommando-Zeile, als auch in den heutigen IDE's wie der Eclipse Warnungen. Die beiden folgenden Abbildungen zeigen dies:

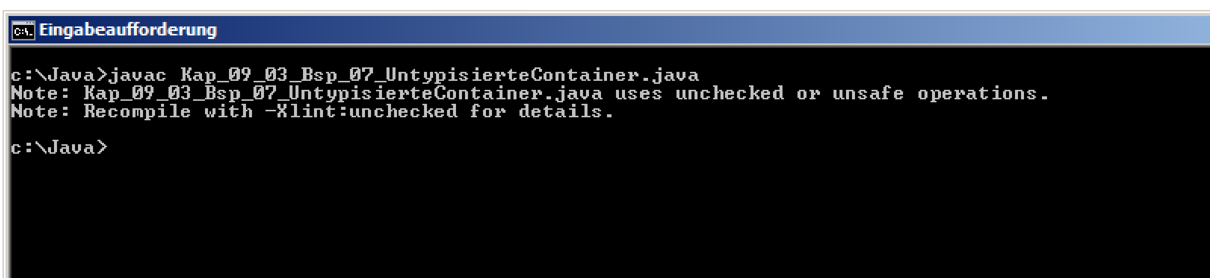

Abb. 9-2 : Warnungen des Java-Compilers wegen der Nutzung untypisierter Container

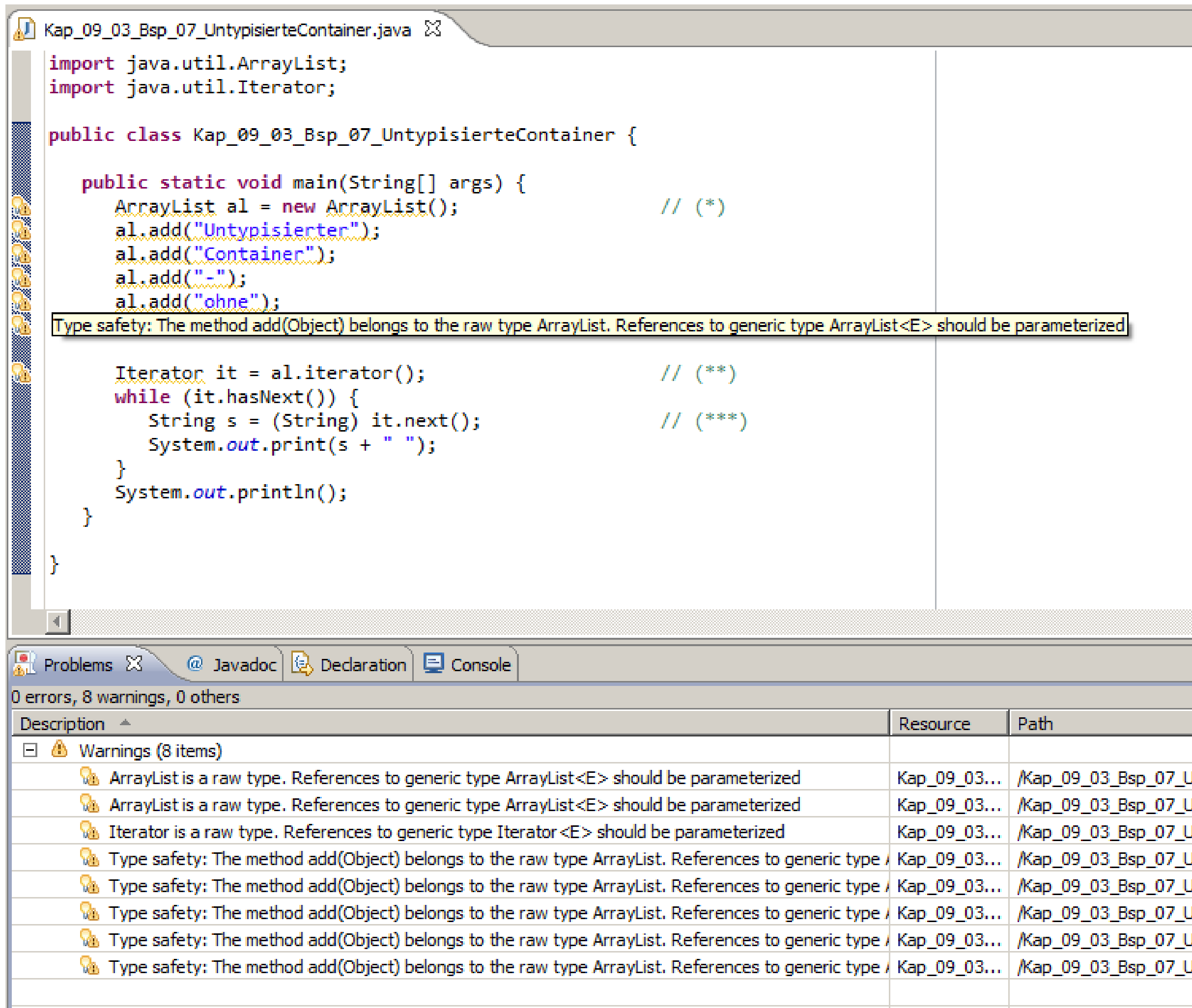

Abb. 9-3 : Warnungen der Eclipse 3.7.2 wegen der Nutzung untypisierter Container

Das freut mich – sowohl der Java-Compiler als auch die Eclipse sind meiner Meinung: Nutzen Sie keine untypisierten Container.

Wenn Sie untypisierte Container aber doch nutzen müssen, dann unterdrücken Sie diese Warnungen, damit die interessanten Warnungen nicht verdeckt werden. Dies können Sie z.B. mit der seit Java 5 (JDK 1.5) vorhandenen Annotation „@SuppressWarnings“ machen. Die Eclipse kann sie automatisch als Quick-Fix für Sie einfügen. Der neue Quelltext sieht dann so aus:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Kap_09_03_Bsp_07_UntypisierteContainer {

   @SuppressWarnings({ "rawtypes", "unchecked" })
   public static void main(String[] args) {
```

```
      ArrayList al = new ArrayList();                    // (*)
      al.add("Untypisierter");
      al.add("Container");
      al.add("-");
      al.add("ohne");
      al.add("Generics");

      Iterator it = al.iterator();                       // (**)
      while (it.hasNext()) {
         String s = (String) it.next();                  // (***)
         System.out.print(s + " ");
      }
      System.out.println();
   }

}
```

Ausgabe

```
Untypisierter Container - ohne Generics
```

Sowohl der Container als auch der Iterator werden hier ohne Element-Typ angegeben – siehe Zeile (*) und (**). Prinzipiell sieht der Code dadurch einfacher aus, aber er ist nur unsicherer:

- Mit „add" können jetzt beliebige Objekte dem Container hinzugefügt werden – nicht nur Strings.
- Und die Iterator-Funktion „next" (Zeile (***)) gibt das Element jetzt nur als „Object" zurück – und dies muss nun explizit von uns gecastet werden (Angabe des Ziel Typs in runden Klammern vor dem Funktions-Aufruf). Enthält der Container andere Elemente, so wird diese Zeile zur Laufzeit schief gehen und eine Class-Cast Exception werfen.

Das folgende Beispiel zeigt das typische Problem untypisierter Container. Das Problem ist, dass der Fehler nicht vom Compiler erkannt wird, sondern erst zur Laufzeit auftritt – und wenn Sie Pech haben, erst beim Kunden.

```
import java.util.ArrayList;
import java.util.Iterator;

public class Kap_09_03_Bsp_08_LaufzeitFehler {

   @SuppressWarnings({ "rawtypes", "unchecked" })
   public static void main(String[] args) {
      ArrayList al = new ArrayList();
      al.add("Untypisierter");
      al.add("Container");
      al.add("-");
      al.add(new StringBuilder());              // Achtung - kein String
      al.add("Generics");

      Iterator it = al.iterator();
      while (it.hasNext()) {
         String s = (String) it.next();         // Laufzeit-Fehler beim
         System.out.print(s + " ");             // vierten Durchlauf
      }
      System.out.println();
   }

}
```

Ausgabe

```
Untypisierter Container -
Exception in thread "main" java.lang.ClassCastException:
java.lang.StringBuilder cannot be cast to java.lang.String
  at Kap_09_03_Bsp_08_LaufzeitFehler.main
     (Kap_09_03_Bsp_08_LaufzeitFehler.java:17)
```

Hinweis – die Container werden in den folgenden Container-Kapiteln nur typisiert vorgestellt, auch wenn auch „untypisiert" möglich wäre. Bevorzugen Sie aber, wann immer möglich, die typisierte Variante. In allen Beispielen und Musterlösungen kommen daher auch nur die typisierten Varianten vor.

9.3.3 Container

Alle Container können hier nicht annähernd vorgestellt werden – dazu gibt zu viele in der Java Bibliothek. Stattdessen beschränken wir uns hier auf die wichtigsten Container:

- java.util.ArrayList – Dynamisches Array (Kapitel 9.3.4)
- java.util.LinkedList – doppelt verkettete Liste (Kapitel 9.3.5)
- java.util.TreeSet – Sortierte Menge (Kapitel 9.3.6)
- java.util.HashSet – Unsortierte gehashte Menge (Kapitel 9.3.7)
- java.util.TreeMap – Sortierter assoziativer Container (Kapitel 9.3.8)
- java.util.HashMap – Unsortierter gehashter assoziativer Container (Kapitel 9.3.9)

Alle Container sind intern typlose Container, d.h. sie können alle Arten von Typen, die nicht elementar sind, aufnehmen – d.h. alle Objekte. Dies hat manchmal Vorteile, führt aber in der Praxis häufig zu Typ-Fehlern, die erst zur Laufzeit auffallen (vergleiche vorhergehendes Kapitel 9.3.2). Daher nutzen Sie bitte immer die typisierte Variante.

Hinweis – um die Klassen ArrayList, TreeMap, Iterator, usw. benutzen zu können, müssen die entsprechenden „import" Anweisungen am Anfang des Quelltextes (nach der package-Anweisung und vor der Klassen-Definition) stehen – siehe Beispiele. Weitere Informationen zu Packages und Imports finden Sie im Kapitel über Packages.

Achtung – alle Java Container können nur Objekte aufnehmen, und **keine** elementaren Datentypen. Einfache Bool-, Zeichen-, Integer- oder Fließkomma-Werte können Sie also nicht direkt in Java Containern speichern. Sie müssen solche Werte in spezielle Wrapper-Klassen einschliessen – siehe Kapitel 9.5. Seit dem JDK 1.5 geschieht dieses Wrappen automatisch, dieses Java Feature nennt sich „Auto-Boxing" – siehe Kapitel 9.5.1. Lassen Sie sich aber nicht täuschen – Java Container können weiterhin keine elementaren Datentypen aufnehmen. Der Effekt hat nichts an der internen Vorgehensweise und Implementierung geändert.

9.3.4 ArrayList

Die ArrayList ein dynamisches Array, d.h. ein Container bei dem die Elemente direkt hintereinander im Speicher liegen und problemlos am Ende angefügt werden können. Ausserdem ist er ein Beispiel für einen sequentiellen Container. Das ist ein Container, in dem die Objekte sequentiell hintereinander liegen (z.B. in der Reihenfolge des Einfügens) und der Container hierbei keinerlei Einfluss auf die Reihenfolge der Objekte nimmt.

todo

```
import java.util.ArrayList;
```

```
import java.util.Iterator;

public class Appl {

   public static void main(String[] args) {
      ArrayList<String> al = new ArrayList<>();
      al.add("Hallo");
      al.add("wunderbarer");
      al.add("Java");
      al.add("Kurs");

      System.out.println("Der Container enthaelt " + al.size() + " Objekte");

      // Neue JDK 1.5 For-Schleife
      for (String s : al) {
         System.out.print(s + " ");
      }
      System.out.println();

      // Iterator-Loesung mit For-Schleife
      for (Iterator<String> it = al.iterator(); it.hasNext();) {
         String s = it.next();
         System.out.print(s + " ");
      }
      System.out.println();

      // Iterator-Loesung mit While-Schleife
      Iterator<String> it = al.iterator();
      while (it.hasNext()) {
         String s = it.next();
         System.out.print(s + " ");
      }
      System.out.println();
   }

}
```

Ausgabe

```
Der Container enthaelt 4 Objekte
Hallo wunderbarer Java Kurs
Hallo wunderbarer Java Kurs
Hallo wunderbarer Java Kurs
```

9.3.5 LinkedList

Die LinkedList ist eine doppelt verkette Liste. Die Elemente sind frei im Speicher verteilt, und untereinander verkettet. Sie verhält sich von der Schnittstelle ähnlich der ArrayList. Wahlfreier Zugriff ist aber viel ineffizienter, dafür ist das Einfügen und Löschen an beliebiger Stelle bei der Liste effizient.

```
import java.util.Iterator;
import java.util.LinkedList;

public class Appl {

   public static void main(String[] args) {
      LinkedList<String> ll = new LinkedList<>();
      ll.add("Hallo");
      ll.add("wunderbarer");
      ll.add("Java");
      ll.add("Kurs");

      System.out.println("Der Container enthaelt " + ll.size() + " Objekte");

      // Neue JDK 1.5 For-Schleife
      for (String s : ll) {
```

```
            System.out.print(s + " ");
        }
        System.out.println();

        // Iterator-Loesung mit For-Schleife
        for (Iterator<String> it = ll.iterator(); it.hasNext();) {
            String s = it.next();
            System.out.print(s + " ");
        }
        System.out.println();

        // Iterator-Loesung mit While-Schleife
        Iterator<String> it = ll.iterator();
        while (it.hasNext()) {
            String s = it.next();
            System.out.print(s + " ");
        }
        System.out.println();
    }

}
```

```
Ausgabe
Der Container enthaelt 4 Objekte
Hallo wunderbarer Java Kurs
Hallo wunderbarer Java Kurs
Hallo wunderbarer Java Kurs
```

9.3.6 TreeSet

Im TreeSet werden die Elemente im Container in Form eines binären Baums angeordnet, daher jedes Element hat bis zu zwei Nachfolger. Dabei werden die kleineren Elemente (bezogen auf den Wert des aktuellen Nodes) links, und die größeren Elemente rechts angeordnet. Dadurch entsteht eine implizite Ordnung im Container, die das folgende Beispiel auch zeigt. Damit ändert der Container aber auch die Reihenfolge der Elemente im Container. Ein Set ist kein sequentieller Container.

Zusätzlich ist ein Set auch eine mathematische Menge. Ein Set kann daher kein Element mehrfach aufnehmen. Ein erneutes Einfügen eines schon vorhandenen Elements wird ignoriert.

Hauptvorteile des Sets sind der Menge-Charakter, die implizite Ordnung, und ein sehr schnelles Suchen auch in großen Mengen.

```
import java.util.Iterator;
import java.util.TreeSet;

public class Appl {

    public static void main(String[] args) {
        TreeSet<String> ts = new TreeSet< >();
        ts.add("Detlef");
        ts.add("Edgar");
        ts.add("Silke");
        ts.add("Martina");
        ts.add("Aaron");

        System.out.println("Der Container enthaelt " + ts.size() + " Objekte");

        boolean exists = ts.contains("Detlef");
        System.out.println("Detlef ist im TreeSet vorhanden: " + exists);

        exists = ts.contains("Hans");
```

```
      System.out.println("Hans ist im TreeSet vorhanden: " + exists);

      // Neue JDK 1.5 For-Schleife
      for (String s : ts) {
         System.out.print(s + " ");
      }
      System.out.println();

      // Iterator-Loesung mit For-Schleife
      for (Iterator<String> it = ts.iterator(); it.hasNext();) {
         String s = it.next();
         System.out.print(s + " ");
      }
      System.out.println();

      // Iterator-Loesung mit While-Schleife
      Iterator<String> it = ts.iterator();
      while (it.hasNext()) {
         String s = it.next();
         System.out.print(s + " ");
      }
      System.out.println();
   }
}
```

```
Ausgabe
Der Container enthaelt 5 Objekte
Detlef ist im TreeSet vorhanden: true
Hans ist im TreeSet vorhanden: false
Aaron Detlef Edgar Martina Silke
Aaron Detlef Edgar Martina Silke
Aaron Detlef Edgar Martina Silke
```

9.3.7 HashSet

Auch das HashSet ist ein Set wie das TreeSet. Daher es ist eine mathematische Menge. Die Elemente werden im Container aber mit einem sogenannten Hashing-Verfahren angeordnet. Von außen sieht die Reihenfolge vollkommen zufällig aus. Der Vorteil des Hashing-Verfahrens ist eine extrem gute Performance beim Suchen, die auch bei größeren Containern nicht ansteigt.

```
import java.util.HashSet;
import java.util.Iterator;

public class Appl {

   public static void main(String[] args) {
      HashSet<String> hs = new HashSet<>();
      hs.add("Detlef");
      hs.add("Edgar");
      hs.add("Silke");
      hs.add("Martina");
      hs.add("Aaron");

      System.out.println("Der Container enthaelt " + hs.size() + " Objekte");

      boolean exists = hs.contains("Detlef");
      System.out.println("Detlef ist im HashSet vorhanden: " + exists);

      exists = hs.contains("Hans");
      System.out.println("Hans ist im HashSet vorhanden: " + exists);

      // Neue JDK 1.5 For-Schleife
      for (String s : hs) {
         System.out.print(s + " ");
      }
```

```
      System.out.println();

      // Iterator-Loesung mit For-Schleife
      for (Iterator<String> it = hs.iterator(); it.hasNext();) {
         String s = it.next();
         System.out.print(s + " ");
      }
      System.out.println();

      // Iterator-Loesung mit While-Schleife
      Iterator<String> it = hs.iterator();
      while (it.hasNext()) {
         String s = it.next();
         System.out.print(s + " ");
      }
      System.out.println();
   }
}
```

```
Mögliche Ausgabe (Die Reihenfolge im Hash-Container ist undefiniert)
Der Container enthaelt 5 Objekte
Detlef ist im HashSet vorhanden: true
Hans ist im HashSet vorhanden: false
Aaron Martina Edgar Silke Detlef
Aaron Martina Edgar Silke Detlef
Aaron Martina Edgar Silke Detlef
```

9.3.8 TreeMap

Maps sind assoziative Container. Bei einem assoziativen Container werden Schlüssel/Wert Paare gespeichert, d.h. dass eigentliche Objekt wird über einen Schlüssel referenziert. Ein typisches Beispiel ist ein KFZ-Kennzeichen als Schlüssel, das das eigentliche Fahrzeug eindeutig referenziert. Oder die Matrikel-Nummer eines Studenten, die den Studenten mit allen Informationen referenziert.

Ein Beispiel dafür ist der binäre Baum „TreeMap“, der außerdem noch implizit die Werte nach dem Schlüssel sortiert. Die TreeMap hat das grundsätzliche Verhalten und die Vorteile eines TreeSets, speichert aber anstatt eines einfachen Wertes ein Schlüssel/Wert Paar.

```
import java.util.Iterator;
import java.util.Map.Entry;
import java.util.TreeMap;

public class Appl {

   public static void main(String[] args) {
      TreeMap<String, String> tm = new TreeMap<>();
      tm.put("Detlef", "1234");
      tm.put("Edgar", "5678");
      tm.put("Silke", "248");
      tm.put("Martina", "999");
      tm.put("Aaron", "646");

      System.out.println("Der Container enthaelt " + tm.size() + " Objekte");

      String s = tm.get("Edgar");
      if (s != null)
         System.out.println("Edgar hat die Nummer: " + s);
      else
         System.out.println("Edgar ist nicht in der TreeMap");

      s = tm.get("Hans");
      if (s != null)
```

```
         System.out.println("Hans hat die Nummer: " + s);
      else
         System.out.println("Hans ist nicht in der TreeMap");

      // Neue JDK 1.5 For-Schleife
      for (Entry<String, String> e : tm.entrySet()) {
         String key = e.getKey();
         String val = e.getValue();
         System.out.print(key + " => " + val + ", ");
      }
      System.out.println();

      // Iterator-Loesung mit For-Schleife
      for (Iterator<Entry<String, String>> it = tm.entrySet().iterator();
         it.hasNext();) {
         Entry<String, String> e = it.next();
         String key = e.getKey();
         String val = e.getValue();
         System.out.print(key + " => " + val + ", ");
      }
      System.out.println();

      // Iterator-Loesung mit While-Schleife
      Iterator<Entry<String, String>> it = tm.entrySet().iterator();
      while (it.hasNext()) {
         Entry<String, String> e = it.next();
         String key = e.getKey();
         String val = e.getValue();
         System.out.print(key + " => " + val + ", ");
      }
      System.out.println();

   }
}
```

```
Ausgabe
Der Container enthaelt 5 Objekte
Edgar hat die Nummer: 5678
Hans ist nicht in der TreeMap
Aaron => 646, Detlef => 1234, Edgar => 5678, Martina => 999, Silke => 248,
Aaron => 646, Detlef => 1234, Edgar => 5678, Martina => 999, Silke => 248,
Aaron => 646, Detlef => 1234, Edgar => 5678, Martina => 999, Silke => 248,
```

9.3.9 HashMap

Eine HashMap ist eine Map auf Basis des HashSets, nur enthält es Schlüssel/Wert Paare.

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map.Entry;

public class Appl {

   public static void main(String[] args) {
      HashMap<String, String> hm = new HashMap<>();
      hm.put("Detlef", "1234");
      hm.put("Edgar", "5678");
      hm.put("Silke", "248");
      hm.put("Martina", "999");
      hm.put("Aaron", "646");

      System.out.println("Der Container enthaelt " + hm.size() + " Objekte");

      String s = hm.get("Edgar");
      if (s != null)
         System.out.println("Edgar hat die Nummer: " + s);
      else
         System.out.println("Edgar ist nicht in der HashMap");
```

```
      s = hm.get("Hans");
      if (s != null)
         System.out.println("Hans hat die Nummer: " + s);
      else
         System.out.println("Hans ist nicht in der HashMap");

      // Neue JDK 1.5 For-Schleife
      for (Entry<String, String> e : hm.entrySet()) {
         String key = e.getKey();
         String val = e.getValue();
         System.out.print(key + " => " + val + ", ");
      }
      System.out.println();

      // Iterator-Loesung mit For-Schleife
      for (Iterator<Entry<String, String>> it = hm.entrySet().iterator();
         it.hasNext();) {
         Entry<String, String> e = it.next();
         String key = e.getKey();
         String val = e.getValue();
         System.out.print(key + " => " + val + ", ");
      }
      System.out.println();

      // Iterator-Loesung mit While-Schleife
      Iterator<Entry<String, String>> it = hm.entrySet().iterator();
      while (it.hasNext()) {
         Entry<String, String> e = it.next();
         String key = e.getKey();
         String val = e.getValue();
         System.out.print(key + " => " + val + ", ");
      }
      System.out.println();

   }
}
```

```
Mögliche Ausgabe (Die Reihenfolge im Hash-Container ist undefiniert)
Der Container enthaelt 5 Objekte
Edgar hat die Nummer: 5678
Hans ist nicht in der HashMap
Aaron => 646, Martina => 999, Edgar => 5678, Silke => 248, Detlef => 1234,
Aaron => 646, Martina => 999, Edgar => 5678, Silke => 248, Detlef => 1234,
Aaron => 646, Martina => 999, Edgar => 5678, Silke => 248, Detlef => 1234,
```

9.4 Stream-API und Lambdas

Im letzten Kapitel haben wir die wichtigsten Container der Java Bibliothek, und ihre Eigenschaften, kennen gelernt. Container sind für das Programmieren (in jeder Programmiersprache) sehr wichtig, da man immer Objekte speichern muss. Nach den elementaren Datentypen, und den Klassen für Texte, sind die Container die wichtigsten und am häufigsten benutzten Klassen.

Will man auf Mengen von Elementen arbeiten (daher in den meisten Fällen auf Containern), so programmieren Java Einsteiger meistens Schleifen, mit denen sie über die Container laufen und die Elemente bearbeiten. Seit Java 8 (JDK 1.8) bietet Java funktionale Mittel, um auf Mengen zu arbeiten – die Stream-API und Lambda Ausdrücke. Um beides im Detail zu verstehen, fehlt uns hier noch einiges an Wissen – und sprengt in der Gesamtheit leider auch den Zeitrahmen der Vorlesung. Trotzdem möchte ich hier an einigen Beispielen die Stream-API und Lambdas einführen. Wenn Sie nach der Vorlesung weiterhin Java programmieren, so

sollten Sie sich näher damit auseinandersetzen. Das Arbeiten auf Mengen wird in modernem Java fast nur noch hiermit gemacht – händische Schleifen sind kein guter Stil.

Alle Beispiele basieren auf einer „ArrayList“ als Container. Dies ist aber nicht notwendig – alle Container unterstützen die Stream-API. ArrayList ist einfach ein einfacher und problemloser Container – von daher habe ich ihn hier bei allen Beispielen genutzt. An die Stream-API kommt man bei allen Containern durch Aufruf der Funktion „stream()“.

9.4.1 Stream-API Beispiel 1

Im ersten Beispiel erzeugen wir eine ArrayList von Strings. Dann laufen wir auf drei verschiedene Arten über die ArrayList und geben die Strings aus.

1. Eine normale händische Schleife – dazu gibt es nicht mehr zu sagen.

```
for (String s : l) {
   System.out.print(s + ' ');
}
System.out.println();
```

2. Nutzung der Stream-API und eines Lambda-Ausdrucks

```
l.stream().forEach(s -> System.out.print(s + ' '));
System.out.println();
```

Über die Funktion „stream()“ auf dem Container erhält man Zugriff auf die Stream-API. Sie stellt u.a. die Funktion „forEach“ zur Verfügung, die die übergebene Funktionalität auf jedem Element der Menge ausführt. Diese auf jedem Element auszuführende Funktionalität implementieren wir in Java (seit Java 8) meistens als Lambda-Ausdruck. Ein Lambda-Ausdruck ist ein Stück Code, der nicht direkt ausgeführt wird, sondern in die Funktion übergeben wird. In Java erkennen wir einen Lambda-Ausdruck an dem Pfeil „->“. Vor dem Pfeil werden die Parameter des Lambdas definiert, ohne dass wir dabei den Typ angeben müssen. Dieser wird vom Java Compiler automatisch deduziert – im Beispiel ist das der Parameter „s“, der als String erkannt wird. Nach dem Pfeil „->“ folgt der eigentlich Code, der dann die Parameter nutzen kann.

Die folgende Code-Zeile sagt also aus: „Führe für jedes Element im Container „l“ den Code „System.out.print(s + ' ') aus“.

```
l.stream().forEach(s -> System.out.print(s + ' '));
```

Was bei einem so einfachen Beispiel vielleicht noch nicht nach einer großen Vereinfachung aussieht, ist bei komplexeren Beispielen dagegen sehr hilfreich. Wir werden das in den folgenden drei Beispielen gleich sehen.

3. Nutzung der Stream-API und einer Methoden-Referenz

```
l.stream().forEach(System.out::println);
```

Will man nur eine einzelne Funktion für jedes Element der Menge aufrufen, so kann man auch nur die Funktion selber übergeben. Beachten Sie bitte, dass hier die Funktion durch zwei Doppelpunkte „::“, und nicht mit einem einzelnen Punkt von den davorliegenden Symbolen abgesetzt ist. Man nennt dies in Java eine Methoden-Referenz. Es ist eine Kurz-Schreibweise für den folgenden Code:

```
l.stream().forEach(s -> System.out.println(s));
```

Das gesamte Beispiel mit allen 3 „Schleifen“ Variationen sieht dann so aus:

```
import java.util.ArrayList;

public class Streams01 {

   public static void main(String[] args) {
      ArrayList<String> l = new ArrayList<>();
      l.add("Java");
      l.add("ist");
      l.add("eine");
      l.add("super");
      l.add("Programmiersprache");

      // Normale haendische Schleife
      for (String s : l) {
         System.out.print(s + ' ');
      }
      System.out.println();

      // Nutzung der Stream-API und eines Lambda-Ausdrucks
      l.stream().forEach(s -> System.out.print(s + ' '));
      System.out.println();

      // Nutzung der Stream-API und einer Methoden-Referenz
      l.stream().forEach(System.out::println);
   }

}
```

```
Ausgabe
Java ist eine super Programmiersprache
Java ist eine super Programmiersprache
Java
ist
eine
super
Programmiersprache
```

9.4.2 Stream-API Beispiel 2

Das zweite Beispiel erzeugt eine ArrayList von 50 Integern mit halbwegs wilden Werten. Integer ist eine Klasse für „int“ Werte, die im nächsten Kapitel detaillierter vorgestellt wird. Da Container nur Referenzen enthalten, sind die elementaren Datentypen als wertbasierte Typen nicht für Container möglich. Darum müssen wir den Umweg über eine solche Hilfs-Klasse nehmen.

Wir wollen dann alle Werte, die kleiner als 4 sind, ausgeben. Dazu setzen wir auf dem Stream die Funktion „filter“ ein, die Elemente ausfiltern kann. Die Bedingung implementieren wir als Lambda-Ausdruck.

Um die einzelnen Schritte auf der Stream-API explizit zu sehen, nutze ich für jeden Schritt eine lokale Variable, in der ich das Ergebnis zwischenspeichere. In der Praxis würde man dies nicht machen, und damit den Code übersichtlicher gestalten – siehe die nächsten beiden Beispiele.

```
import java.util.ArrayList;
import java.util.stream.Stream;

public class Streams02 {

   public static void main(String[] args) {
      ArrayList<Integer> myList = new ArrayList<>();
      for (int i=0; i<100; i++) {
         myList.add((i*(i+1)) % 13);             // Erzeugt halbwegs wilde Werte
      }

      Stream<Integer> sequentialStream = myList.stream();
      Stream<Integer> numbers = sequentialStream.filter(value -> value < 4);
      numbers.forEach(value -> System.out.print(value +  " "));
   }

}
```

```
Ausgabe
0 2 3 2 0 0 2 3 2 0 0 2 3 2 0 0 2 3
```

9.4.3 Stream-API Beispiel 3

Das dritte Beispiel transformiert alle Strings des Streams in Großbuchstaben und gibt diese dann aus. Dabei wird der Stream hier direkt mit der Funktion „Stream.of“ erzeugt, ohne den Umweg über einen Container zu gehen. Für die Transformation wird die Stream-API Funktion „map“ genutzt – die eigentliche Transformation implementieren wir wieder mit einem Lambda-Ausdruck, der die Funktion „toUpperCase“ für Strings nutzt. Außerdem verbinden wir die einzelnen Stream-API Funktionen direkt miteinander, ohne lokale Variablen nutzen zu müssen.

```
import java.util.stream.Stream;

public class Streams03 {

   public static void main(String[] args) {
      Stream.of("aBc", "d", "ef")
         .map(s -> { return s.toUpperCase();})
         .forEach(s -> System.out.print(s + " ")
      );
   }

}
```

```
Ausgabe
ABC D EF
```

9.4.4 Stream-API Beispiel 4

Das letzte Beispiel führt als neue Funktion der Stream-API „sort“ ein, welche die Elemente sortiert – der Rest sind die aus den letzten drei Beispielen bekannten Elemente. Wir erzeugen eine ArrayList von Strings. Zuerst werden die Strings getrimmt (alle führenden und hinten stehenden Whitespaces entfernt), die leeren Strings werden ausgefiltert, und alle übrigen in

Zahlen verwandelt. Diese Zahlen werden quadriert, und die Resultate sortiert. Zum Schluss wird das sortierte Resultat ausgegeben.

```
import java.util.ArrayList;

public class Streams04 {

   public static void main(String[] args) {
      ArrayList<String> l = new ArrayList<>();
      l.add("4");
      l.add(" 2");
      l.add("");
      l.add(" 1 ");
      l.add(" 5      ");
      l.add("    ");
      l.add("3 ");

      l.stream()
       .map(String::trim)
       .filter(s -> !s.isEmpty())
       .map(Integer::parseInt)
       .map(v -> v*v)
       .sorted()
       .forEach(val -> System.out.print(val + " "));
   }

}
```

```
Ausgabe
1 4 9 16 25
```

Ich hoffe, Sie sehen, dass die Nutzung der Stream-API und von Lambdas das Arbeiten auf Mengen sehr einfach und leserlich gestaltet. Tiefer werden im Rahmen der Vorlesung leider nicht in diese Themen einsteigen können.

9.5 Wrapper Klassen

In Java gibt es für alle elementaren Datentypen (inkl. „void“) sogenannte Wrapper-Klassen. Sie haben drei Aufgaben:

- Sie dienen als Wrapper-Klassen für z.B. das Container-Framework – siehe Kapitel 9.3.3 und das folgende Kapitel 9.5.1. Hinweis – die Wrapper-Objekte sind wie Strings unveränderbar („immutable“), d.h. die Werte in den Objekten lassen sich nicht ändern.
- Sie sammeln zu dem Typ gehörige allgemeine Funktionen in Form von Klassen-Funktionen. Z.B. die Funktionen zum Konvertieren, wie „parseInt“.

Elementarer Datentyp	Wrapper-Klasse
Boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

© Detlef Wilkening 2021

void Void

Bis auf die Wrapper-Klassen für „char“ und „int“ haben sie den gleichen Namen wie der jeweilige elementare Datentyp den sie kapseln, beginnen aber groß.

9.5.1 Auto-Boxing

In Kapitel 9.3.3 haben wir gelernt, dass die Container-Klassen keine elementaren Daten-Typen aufnehmen können, sondern nur Objekte. Das lässt uns keine Ruhe, und darum stellen wir diese Aussage hier auf die Probe:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Kap_09_04_Bsp_01_AutoBoxing {

   @SuppressWarnings({ "unchecked", "rawtypes" })
   public static void main(String[] args) {
      ArrayList al = new ArrayList();
      al.add(1);                               // "1" ist ein "int"
      al.add(6);                               // "6" ist ein "int"
      al.add(23);                              // "23" ist ein "int"
      al.add(42);                              // "42" ist ein "int"

      System.out.println("Die Liste enthaelt " + al.size() + " ints");

      Iterator it = al.iterator();
      while (it.hasNext()) {
         int n = (int) it.next();              // Cast in "int"
         System.out.print(n + " ");
      }
   }

}
```

```
Ausgabe
Die Liste enthaelt 4 ints
1 6 23 42
```

Funktioniert doch!? Nun, wenn Sie Kapitel 9.3.3 vollständig in Ruhe lesen, dann finden Sie noch eine weitere Aussage: „Elementare Datentypen müssen in speziellen Wrapper-Klassen (siehe Kapitel 9.5) gewrappt werden. Seit dem JDK 1.5 kann dieses Wrappen implizit geschehen (Auto-Boxing) – dies hat aber nichts an der internen Vorgehensweise und Implementierung geändert“.

Hinweis – das Beispiel arbeitet mit einem untypisierten Container, da eine Typisierung auf „int“ zu einem Compiler-Fehler geführt hätte, und wir mit der korrekten Typisierung auf „Integer“ die „Überraschung“ viel kleiner gewesen wäre.

Intern passiert also dass im folgenden Beispiel explizit programmierte – nur das Sie es seit dem JDK 1.5 nicht sehen, da der Compiler es automatisch für Sie macht. Aber Sie dürfen es natürlich immer noch selber machen, bzw. bei einem älteren JDK (vor 1.5) müssen Sie es sogar.

```
import java.util.ArrayList;
import java.util.Iterator;
```

```
public class Appl {

   public static void main(String[] args) {
      ArrayList<Integer> al = new ArrayList<>();
      al.add(Integer.valueOf(1));
      al.add(Integer.valueOf(6));
      al.add(Integer.valueOf(23));
      al.add(Integer.valueOf(42));

      System.out.println("Die Liste enthaelt " + al.size() + " Integer");

      Iterator<Integer> it = al.iterator();
      while (it.hasNext()) {
         Integer i = (Integer) it.next();
         int n = i.intValue();
         System.out.print(n + " ");
      }
   }

}
```

```
Ausgabe
Die Liste enthaelt 4 ints
1 6 23 42
```

- Beim Einfügen in den Container wird der elementare Datentyp (hier „int“) in eine Wrapper-Klasse (hier „Integer“) gewrappt („geboxt“). Beim Auto-Boxing geschieht dies automatisch.
- Möglicherweise haben Sie beim Einfügen „new Integer(x)“ statt „Integer.valueOf(x)“ erwartet – immerhin müssen alle Objekte mit „new“ erzeugt werden. Im Prinzip würde „new Integer(x)“ hier auch funktionieren, aber die Klassen-Funktion „valueOf“ gibt auch ein entsprechendes Java Integer-Objekt zurück – kann aber schneller sein. Für genaue Details schauen Sie bitte in die Java Referenz-Dokumentation.
- Beim Wandeln des Integer-Objekts in einen „int“ muß die Element-Funktion „intValue" genutzt werden. Beim Auto-Boxing geschieht auch dies automatisch.
- Die ArrayList ist in diesem Beispiel auf „Integer“ typisiert – das ist die passende Typisierung um via Auto-Boxing „int“s aufzunehmen.

Intern bleibt es aber so, dass Java Container keine elementaren Datentypen aufnehmen können. Aber seit dem JDK 1.5 hilft uns die Sprache hier ganz automatisch.

9.6 Zufalls-Zahlen

Um Zufalls-Zahlen zu erzeugen gibt es im Package „java.util“ die Klasse „Random“. Man erzeugt sich ein Objekt der Klasse „Random“ und kann sich von diesem Objekt Zufalls-Zahlen erzeugen lassen. Es existieren Element-Funktionen zur Erzeugung von Zufalls-Zahlen für die wichtigsten Typen („boolean“, „int“, „long“, „float“ und „double“) und einige weitere Anwendungsfälle wie z.B. Byte-Arrays.

Die einfachste Element-Funktion ist „nextInt“, die einen Int-Paramter bekommt und dann Zufalls-Zahlen zwischen „0“ (inkl.) und dem übergebenden Argument (exkl.) erzeugt. Im folgenden Beispiel also Zahlen von „0-9“.

```
import java.util.Random;

public class Kap_09_05_Bsp_01_ZufallsZahlen {
```

```
   public static void main(String[] args) {
      Random rnd = new Random();
      for (int i=0; i<20; i++) {
         System.out.print(rnd.nextInt(10) + " ");
      }
   }

}
```

```
mögliche Ausgabe
2 2 5 3 7 0 3 6 4 5 0 7 4 0 3 8 8 7 6 8
```

Hinweis – vergessen Sie nicht den notwendigen Import von „java.util.Random".

Benötigen Sie Zufalls-Zahlen in einem anderen Bereich (d.h. nicht von „0" (inkl.) und dem übergebenden Argument (exkl.)), dann verschieben Sie die Zahlen einfach durch eine Addition oder Subtraktion. Um z.B. einen Würfel mit den Zahlen von „1-6" zu simulieren, benötigen Sie Zahlen von „0-5" die Sie durch eine Addition mit „1" in den gewünschten Bereich verschieben.

```
import java.util.Random;

public class Kap_09_05_Bsp_02_Wuerfel {

   public static void main(String[] args) {
      Random rnd = new Random();
      for (int i = 0; i < 10; i++) {
         System.out.print(rnd.nextInt(6) + 1 + " ");
      }
   }

}
```

```
mögliche Ausgabe
3 4 1 6 5 3 3 2 6 2
```

Hinweise:

- Benötigt man reproduzierbare Zufalls-Zahlen, so kann man ein Random-Objekt auch mit einem Seed-Parameter konstruieren, oder bei einem vorhandenen Random-Objekt den Seed mit der Funktion „setSeed" setzen.
- Double-Zufalls-Zahlen kann man sich auch mit der Klassen-Funktion „random" der Klasse „java.lang.Math" erzeugen lassen. Die genauen Unterschiede zwischen dieser Funktion und „nextDouble" aus „Random" finden Sie in der Java Referenz-Dokumentation.

9.7 Datum und Uhrzeit

Für Datum und Uhrzeit Funktionen stehen u.a. folgende Klassen zur Verfügung:

- java.util.Date
- java.util.Calendar
- java.text.DateFormat
- java.text.SimpleDateFormat

Wird einfach ein Date Objekt erzeugt ("new Date()"), so enthält dieses Objekt das aktuelle Datum und die aktuelle Zeit. Die Klassen „DateFormat" und „SimpleDateFormat" bieten Methoden zur Formatierung der Wandlung des Datums in einen String.

```
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

public class Kap_09_06_Bsp_01_Datum {

   public static void main(String[] args) {
      Date now = new Date();

      DateFormat df = DateFormat.getDateInstance();
      System.out.println(df.format(now));

      df = DateFormat.getDateInstance(DateFormat.FULL);
      System.out.println(df.format(now));

      df = DateFormat.getTimeInstance();
      System.out.println(df.format(now));

      df = DateFormat.getTimeInstance(DateFormat.FULL);
      System.out.println(df.format(now));

      df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL);
      System.out.println(df.format(now));

      df = new SimpleDateFormat("d.M.yyyy");
      System.out.println(df.format(now));
   }
}
```

```
mögliche Ausgabe
06.05.2012
Sonntag, 6. Mai 2012
01:05:39
01:05 Uhr MESZ
Sonntag, 6. Mai 2012 01:05 Uhr MESZ
6.5.2012
```

- Die Definition eines Ausgabe-Formats mit „DateFormat“ arbeitet defaultmäßig mit der aktuellen Länderkennung - dies lässt sich natürlich ändern.
- „DateFormat“ kann auch Strings parsen – Element-Funktion „parse“.
- Mit der Klasse „Calendar“ kann Datums- und Uhrzeit-Arithmetik mit Date-Objekten durchgeführt werden, und es können viele weitere Informationen gewonnen werden.
- Achtung – viele Funktionen in „Date“ sind „deprecated“ und durch Funktionen der Klasse „Calendar“ ersetzt worden.

Hinweis – mit dem JDK 1.8 ist eine neue Date-Time Bibliothek in Java eingeführt worden. Diese ist viel durchdachter und leistungsfähiger als die alte hier kurz angesprochene Bibliothek. In realen Projekten sollten Sie besser diese nutzen.

9.8 Datei- und Verzeichnis-Handling

Die Klasse „java.io.File“ repräsentiert eine Datei oder ein Verzeichnis, und sie bietet viele Funktionen um Dateien und Verzeichnisse zu manipulieren oder Informationen über sie zu erlangen. Hier ein kleiner Auszug aus der Schnittstelle der Klasse „File“:

File(String fullName)	Erzeugt ein File Objekt für das Element des Datei-Systems mit dem entsprechenden Namen. Der Name ist vollständig inkl.

	Pfad.
File(String path, String name)	Erzeugt ein File Objekt für das Element des Datei-Systems mit dem entsprechenden Namen. Der Name wird hierbei getrennt in Pfad und eigentlichem Namen übergeben. Der Vorteil ist hier, dass der Benutzer sich nicht um den Trenner im String kümmern muß, der ja plattform-spezifisch ist.
boolean exists()	Gibt zurück, ob die Datei oder das Verzeichnis existiert.
String getAbsolutePath()	Gibt den Namen (inkl. vollständigem absolutem Pfad) des Verzeichnisses oder der Datei zurück.
boolean isDirectory()	Gibt zurück, ob das File-Objekt ein Verzeichnis referenziert.
boolean isFile()	Gibt zurück, ob das File-Objekt eine Datei referenziert.
long length()	Gibt bei einer Datei die Größe der referenzierten Datei zurück.
String[ ] list()	Gibt bei einem Verzeichnis die Namen der Elemente im Verzeichnis (Verzeichnisse und Dateien) als Array von Strings zurück.

Als erstes Beispiel ein kleines Programm, dass überprüft ob der auf der Kommandozeile übergebene Name im Datei-System existiert, und ob es eine Datei oder ein Verzeichnis ist. Wenn es eine Datei ist, so wird noch die Größe der Datei ausgegeben.

```
import java.io.File;

public class Kap_09_07_Bsp_01_File {

   public static void main(String[] args) {
      if (args.length!=1) {
         System.out.println("Fehler - es wird ein Datei-Name erwartet");
         return;
      }

      String name = args[0];
      File file = new File(name);

      if (!file.exists()) {
         System.out.println("Element \"" + name + "\" existiert nicht");
         return;
      }

      if (file.isFile()) {
         String len = "" + file.length();
         System.out.println("\"" + name + "\" hat " + len + " Bytes");
         return;
      }

      if (file.isDirectory()) {
         System.out.println("\"" + name + "\" ist ein Verzeichnis");
         return;
      }

      System.out.println("Element \"" + name + "\" -> unbekannten Typ");
   }

}
```

Mögliche Ausgabe

```
>java Appl
Fehler - es wird ein Datei-Name erwartet
```

```
Mögliche Ausgabe
>java Appl java
Element "java" existiert nicht
```

```
Mögliche Ausgabe
>java Appl Kap_09_07_Bsp_01_File.class
"Kap_09_07_Bsp_01_File.class" hat 1414 Bytes
```

```
Mögliche Ausgabe
>java Appl .
"." ist ein Verzeichnis
```

Hinweise

- Für die Verwendung von plattform-unabhängigen Datei-Namen wird in Java die URI Konvention gewählt, für die es eine eigene Klasse „java.net.URI“ gibt.
- Lesen und Schreiben von Dateien geschieht in Java mit Streams, die mit einem File-Objekt verbunden sind.

9.8.1 Rekursive Datei-Suche

Ein typisches Problem, dass wir nun lösen können, ist die tiefe Suche im Dateisystem z.B. nach einer Datei mit einem bestimmten Namen. Bevor Sie sich an dieses Unternehmen wagen, empfehle ich Ihnen, sich sinnvolle Testdaten zu erzeugen – sprich eine kleine Beispiel Verzeichnis-Struktur. Ich habe es leider erlebt, dass Anfänger ihr Programm zum Test auf ihr Root-Dateisystem wie z.B. „C:\“ losgelassen haben, und sich dann gewundert haben dass ihr Programm ewig läuft, Fehler meldet oder scheinbar einfach nur den Rechner lahm legt. Ist doch auch kein Wunder, oder? Bei unseren heutigen Plattengrößen gibt es da ziemlich viele Verzeichnisse zu durchsuchen und Datei-Namen zu vergleichen, dass der Rechner gut belastet ist. Und mit ziemlicher Sicherheit trifft das Programm unterwegs auf Verzeichnisse, für die es keine Rechte hat – und schon regnet es Exceptions, und damit können wir noch gar nicht gut umgehen.

Lange Rede, kurzer Sinn – machen Sie sich eine kleine Test-Daten-Struktur auf Ihrer Platte. Ich habe das gemacht, unter „C:\JavaDateiSystemBeispiele“ – und so sieht sie bei mir aus. Suchen wollen wir später die drei Dateien mit dem Namen „find.txt“.

```
Test-Verzeichnis-Struktur
C:\
└ 🗁 JavaDateiSystemBeispiele
  └ 🗁 verzeichnis1
  │ └ 🗁 verzeichnis3
  │ │ └ 🗁 verzeichnis4
  │ │ │ └ 🖹 find.txt
  │ │ │ └ 🖹 search.dat
  │ │ │ └ 🖹 test.txt
  │ │ └ 🖹 test.txt
  │ └ 🗁 verzeichnis5
  │ │ └ 🖹 find.txt
  │ │ └ 🖹 test.txt
```

```
|   └ 🗎 daten.dat
|   └ 🗎 test.txt
└ 🗁 verzeichnis2
|   └ 🗎 find.txt
|   └ 🗎 test.txt
└ 🗎 search.dat
└ 🗎 test.txt
```

Hinweis – bitte ignorieren Sie im Augenblick die blau markierten Dateien „search.dat“ und „daten.dat“ – wir benötigen Sie später.

Damit können wir nun loslegen. Der erste Schritt ist einfach. Wir laufen einfach über das Such-Verzeichnis und geben alles aus, was wir finden – getrennt nach Verzeichnissen und Dateien. Einzige Besonderheit: wir geben den Namen vollständig aus, und setzen ihn nicht selber zusammen, sondern lassen ihn uns vom File-Objekt geben – Zeile (*). Hintergrund hierfür sind die unterschiedlichen Datei-Trenner unter z.B. Windows (Backslash „\“) und Linux (Slash „/“), um die wir uns sonst selbst kümmern müssten.

```
import java.io.File;

public class Kap_09_07_Bsp_02_FileSucheRekursivStep1 {

   public static void main(String[] args) {
      File path = new File("C:\\JavaDateiSystemBeispiele");
      String[] filenames = path.list();
      for (String filename : filenames) {
         File file = new File(path, filename);
         String fullfilename = file.getAbsolutePath();          // (*)
         if (file.isDirectory()) {
            System.out.println("Verz.: " + fullfilename);
         } else if (file.isFile()) {
            System.out.println("Datei: " + fullfilename);
         }
      }
   }

}
```

```
Mögliche Ausgabe (andere Reihenfolge möglich – ist nicht definiert)
Datei: C:\JavaDateiSystemBeispiele\search.dat
Datei: C:\JavaDateiSystemBeispiele\test.txt
Verz.: C:\JavaDateiSystemBeispiele\verzeichnis1
Verz.: C:\JavaDateiSystemBeispiele\verzeichnis2
```

Der nächste Schritt ist minimal – wir vergleichen den gefundenen Datei-Namen mit dem Namen, den wir suchen – und damit wir was finden, suchen wir erstmal nach „test.txt“. Wenn die Datei stimmt, dann geben wir sie mit komplettem Pfad aus. Und bei Verzeichnissen machen wir erstmal nix.

```
import java.io.File;

public class Kap_09_07_Bsp_03_FileSucheRekursivStep2 {

   public static void main(String[] args) {
      String searchfile = "test.txt";
      File path = new File("C:\\JavaDateiSystemBeispiele");
      String[] filenames = path.list();
      for (String filename : filenames) {
         File file = new File(path, filename);
         String fullfilename = file.getAbsolutePath();
         if (file.isDirectory()) {
```

```
            // Im Augenblick machen wir hier noch nichts
         } else if (file.isFile()) {
            if (file.getName().equals(searchfile)) {
               System.out.println("-> " + fullfilename);
            }
         }
      }
   }

}
```

Ausgabe
```
-> C:\JavaDateiSystemBeispiele\test.txt
```

Kommen wir nun zum Fall, dass das Verzeichnis selber wieder ein Verzeichnis enthält. In diesem Fall müssen wir nur für dieses Verzeichnis das Gleiche machen: alle Elemente durchlaufen, Dateien vergleichen, bei Verzeichnissen wieder das Gleiche. Das klingt nach Rekursion, wie wir sie in Kapitel 8.7 kennen gelernt haben. Wichtig – dazu müssen wir die Funktionalität in eine Funktion auslagern – sonst können wir sie nicht rekursiv aufrufen. Und mehr machen wir in diesem Schritt auch nicht.

```
import java.io.File;

public class Kap_09_07_Bsp_04_FileSucheRekursivStep3 {

   public static void main(String[] args) {
      String searchpath = "C:\\JavaDateiSystemBeispiele";
      String searchfile = "test.txt";
      searchFile(searchpath, searchfile);
   }

   public static void searchFile(String searchpath, String searchfile) {
      File path = new File(searchpath);
      String[] filenames = path.list();
      for (String filename : filenames) {
         File file = new File(path, filename);
         String fullfilename = file.getAbsolutePath();
         if (file.isDirectory()) {
            // Im Augenblick machen wir hier noch nichts
         } else if (file.isFile()) {
            if (file.getName().equals(searchfile)) {
               System.out.println("-> " + fullfilename);
            }
         }
      }
   }

}
```

Ausgabe
```
-> C:\JavaDateiSystemBeispiele\test.txt
```

Nachdem unsere Verzeichnis-Durchforste-Funktionalität nun in einer Funktion ist, können wir sie jetzt auch aufrufen – und diesmal auch mit dem richtigen Datei-Namen.

```
import java.io.File;

public class Kap_09_07_Bsp_05_FileSucheRekursivStep4 {

   public static void main(String[] args) {
      String searchpath = "C:\\JavaDateiSystemBeispiele";
      String searchfile = "find.txt";
      searchFile(searchpath, searchfile);
   }

   public static void searchFile(String searchpath, String searchfile) {
```

```
      File path = new File(searchpath);
      String[] filenames = path.list();
      for (String filename : filenames) {
         File file = new File(path, filename);
         String fullfilename = file.getAbsolutePath();
         if (file.isDirectory()) {
            searchFile(fullfilename, searchfile);
         } else if (file.isFile()) {
            if (file.getName().equals(searchfile)) {
               System.out.println("-> " + fullfilename);
            }
         }
      }
   }

}
```

```
Mögliche Ausgabe (andere Reihenfolge möglich - ist nicht definiert)
-> C:\JavaDateiSystemBeispiele\verzeichnis1\verzeichnis3\verzeichnis4\find.txt
-> C:\JavaDateiSystemBeispiele\verzeichnis1\verzeichnis5\find.txt
-> C:\JavaDateiSystemBeispiele\verzeichnis2\find.txt
```

Wow – alles funktioniert. So einfach und elegant ist Rekursion. Damit unser Quelltext noch etwas mehr nach richtigem Programm aussieht, integrieren wir noch ein paar Dinge:

- Programm-Name
- Rückmeldung, wo gesucht wird
- Rückmeldung, wonach gesucht wird
- Und eine Sicherheits-Abfrage, falls jemand „searchFile“ nicht mit einem Verzeichnis als ersten Parameter aufruft.

```
import java.io.File;

public class Kap_09_07_Bsp_06_FileSucheRekursiv {

   public static void main(String[] args) {
      System.out.println("Rekursive Datei-Suche mit java.io.File");

      String searchpath = "C:\\JavaDateiSystemBeispiele";
      String searchfile = "find.txt";
      System.out.println("Suche in: " + searchpath);
      System.out.println("Nach: " + searchfile);

      searchFile(searchpath, searchfile);
   }

   public static void searchFile(String searchpath, String searchfile) {
      File path = new File(searchpath);
      if (!path.isDirectory()) {
         System.out.println("Fehler - kein Verzeichnis");
         return;
      }

      String[] filenames = path.list();
      for (String filename : filenames) {
         File file = new File(searchpath, filename);
         String fullfilename = file.getAbsolutePath();
         if (file.isDirectory()) {
            searchFile(fullfilename, searchfile);
         } else if (file.isFile()) {
            if (searchfile.equals(filename)) {
               System.out.println("-> " + fullfilename);
            }
         }
      }
   }
```

```
}
```

```
Mögliche Ausgabe (andere Reihenfolge möglich - ist nicht definiert)
Rekursive Datei-Suche mit java.io.File
Suche in: c:\aaa
Nach: find.txt
-> C:\JavaDateiSystemBeispiele\verzeichnis1\verzeichnis3\verzeichnis4\find.txt
-> C:\\JavaDateiSystemBeispiele\verzeichnis1\verzeichnis5\find.txt
-> C:\JavaDateiSystemBeispiele\verzeichnis2\find.txt
```

Hinweise:

- In einigen Aufgaben sollen Sie dieses Programm zur rekursiven Datei-Suche um weitere Fähigkeiten ergänzen.
- Im nächsten Kapitel wird die rekursive Datei-Suche mit dem JDK 1.7 Package „java.nio.file“ implementiert.

9.8.2 Rekursive Datei-Suche mit „java.nio.file“

Seit dem JDK 1.4 gibt es das Package New-IO „java.nio“ in Java. In Java 7 wurde es um das Unter-Package „java.nio.file“ ergänzt. Hier drin sind viele Klassen mit stark erweiterten Möglichkeiten gegenüber „java.io.file“ enthalten – über Kopieren von Dateien und Verzeichnissen bis hin zu Themen wie User-Rechte auf Datei-Systeme. So leistungsfähig das neue „java.nio.file“ Package auch ist – die Nutzung ist für Einsteiger oft schwer und komplex. Von daher nutzen wir hier nur die File-Klasse – später sollten Sie sich aber ruhig mal an „java.nio.file“ erinnern, falls Sie aufwändigere Aufgaben im Kontext Datei-System lösen müssen.

Als Beispiel, aber ohne jede Erklärung, nochmal die rekursive Datei-Suche – nun aber mit „java.nio.file“.

```
import java.io.File;
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Kap_09_07_Bsp_03_FileSucheRekursivMitNiofile {

   public static void main(String[] args) {
      System.out.println("Rekursive Datei-Suche mit java.nio.file");

      String searchpath = "C:\\JavaDateiSystemBeispiele";
      String searchfile = "find.txt";
      System.out.println("Suche in: " + searchpath);
      System.out.println("Nach: " + searchfile);

      try {
         Path path = Paths.get(searchpath);
         searchFile(path, searchfile);
      } catch (IOException e) {
         System.out.println("Fehler: " + e.getMessage());
      }
   }

   public static void searchFile(Path path, String searchfile)
   throws IOException {
      DirectoryStream<Path> dirStream = Files.newDirectoryStream(path);
      for (Path entry : dirStream) {
```

```
            File file = entry.toFile();
            if (file.isDirectory()) {
               searchFile(entry, searchfile);
            } else if (file.isFile()) {
               if (file.getName().equals(searchfile)) {
                  System.out.println("-> " + file.getAbsolutePath());
               }
            }
         }
      }

}
```

Ausgabe (unter Annahme der obigen Datei-Struktur)

```
Rekursive Datei-Suche mit java.nio.file
Suche in: c:\aaa
Nach: find.txt
-> C:\JavaDateiSystemBeispiele\verzeichnis1\verzeichnis3\verzeichnis4\find.txt
-> C:\JavaDateiSystemBeispiele\verzeichnis1\verzeichnis5\find.txt
-> C:\JavaDateiSystemBeispiele\verzeichnis2\find.txt
```