Die Klasse String

Ein String repräsentiert eine Zeichenkette, weshalb die Konvertierung zwischen einer Liste vom Typ char und dem String verlustfrei ist:

```
char[] zeichenListe = {'M', 'i', 'c', 'h', 'a'};
String micha = new String(zeichenListe);     // char[] -> String
zeichenListe = micha.toCharArray();          // String -> char[]
```

Objekte vom Typ String sind nach der Initialisierung nicht mehr veränderbar.

Erzeugung und Manipulation von Zeichenketten

Mit dem folgenden Beispiel sehen wir zwei weitere Möglichkeiten, Objekte der Klasse String zu erzeugen:

```
String paul  = new String("Paul");
String peter = "Peter";
```

In der ersten Variante sehen wir den üblichen, expliziten Weg, eine Objektreferenz zu deklarieren und entsprechend Speicher bereitzustellen. Die zweite Variante stellt eine spezielle, implizite Möglichkeit in Java dar, Zeichenketten über den sogenannten Stringpool anzulegen.

Den entscheidenden Unterschied beider Varianten werden wir verstehen, wenn wir versuchen, Zeichenketten zu vergleichen.

Vergleich von Zeichenketten

Wenn wir uns daran erinnern, dass wir es auch in diesem Beispiel mit Objektreferenzen zu tun haben, dann erklärt es sich einfach, dass auch Stringvergleiche der Methode equals bedürfen:

```
String a = new String("Hugo");
String b = new String("Hugo");

if (a == b)
   System.out.println("Objektreferenzen sind gleich");
else
   System.out.println("Objektreferenzen sind ungleich");

if (a.equals(b))
   System.out.println("Inhalte sind gleich");
else
   System.out.println("Inhalte sind ungleich");
```

Wenn wir diesen Programmabschnitt ausführen, erhalten wir die folgende, erwartete Ausgabe:

```
Objektreferenzen sind ungleich
Inhalte sind gleich
```

Vergleich von Zeichenketten mit Stringpool

Bei der Erzeugung einer Zeichenketten über die Stringpool-Variante wird zunächst überprüft, ob diese Zeichenkombination bereits in der Vergangenheit des aktuell laufenden Programms erzeugt wurde. Wenn dies der Fall ist, wird nur die gleiche Objektreferenz vergeben, ansonsten die neue Zeichenkette dem Pool hinzugefügt und entsprechend die neue Objektreferenz zurückgegeben. Das Verhalten sehen wir sehr schön in dem folgenden Beispiel:

```
String a = "Hugo";
String b = "Hugo";

... Rest identisch
```

Wenn wir diesen Programmabschnitt ausführen, erhalten wir die folgende, erwartete Ausgabe:

```
Objektreferenzen sind gleich
Inhalte sind gleich
```

Allerdings müssen wir an dieser Stelle auf ein kleines Risiko hinweisen: Es gibt für den Stringpool eine Begrenzung des Speichers, weshalb es bei großen Datenmengen unter Umständen dazu kommen kann, dass der Vergleich nicht mehr das korrekte Resultat liefert.

Manipulation von Zeichenketten

Wir gehen für die folgenden Beispiele von diesen vier definierten Zeichenketten aus:

```
String n1     = "John";
String n2     = "Ronald";
String n3     = "Reuel";
String n4     = "Tolkien";
```

Wenn wir Zeichenketten verknüpfen wollen, können wir das über den Operator + oder die in der Klasse String vorhandene Methode concat vornehmen:

```
String kette1 = n1+" "+n4;
String kette2 = n1.concat(" ") + n4;
String kette3 = (n1.concat(" ")).concat(n4);
```

Die Beispiele zeigen auch, dass sich beide beliebig verknüpfen lassen. Für alle drei Zeichenketten ergibt sich bei der Ausgabe die folgende Zeichenkette:

```
John Tolkien
```

Manipulation von Zeichenketten II

Im Unterschied zu Arrays, bei denen wir ein Attribut length verwendet haben, ist bei Zeichenketten über eine Methode length, die Anzahl der Einträge zu ermitteln:

```
int laenge1 = (new char[] {'a','d','a','m'}).length;  // Attribut
int laenge2 = kette1.length();                         // Funktion
int laenge3 = "Tolkien".length();                      // Funktion
```

Über die Methode substring können wir eine Teilsequenz der Zeichenkette auslesen:

```
String teil = kette1.substring(8, 12);
System.out.println("Teil: >"+teil+"<");
```

Da in kette1 der Inhalt John Tolkien steht, erhalten wir als Teilsequenz von Zeichen 8 bis Zeichen 12 folgende Ausgabe:

```
Teil: >kien<
```

Manipulation von Zeichenketten III

Wenn wir die ersten Buchstaben der zu Beginn des Abschnitts definierten Vornamen abkürzen und mit dem Nachnamen entsprechend verknüpfen:

```
String kombination = n1.substring(0, 1)+". "+
                     n2.substring(0, 1)+". "+
                     n3.substring(0, 1)+". "+
                     n4;
System.out.println("Kombination: "+kombination);
```

Erhalten wir das Kürzel eines berühmten Schriftstellers:

```
J. R. R. Tolkien
```

Live-Coding-Session

Prinzip des Überladens

Oft ist es sinnvoll, eine Methode für verschiedene Eingabetypen zur Verfügung zu stellen. Damit wir nicht jedes Mal einen neuen Methodennamen wählen müssen, gibt es die Möglichkeit des Überladens von Methoden.

Wir können einen Methodennamen mehrfach verwenden, falls sich die Signatur der Methoden eindeutig unterscheiden lässt. Als Signatur definieren wir die Kombination aus Rückgabetyp, Methodenname und Eingabeparameter wobei nur der Typ der Eingabe und nicht die Bezeichnung entscheidend ist. Eine Unterscheidung nur im Rückgabetyp ist dabei nicht zulässig.

Schauen wir uns als Beispiel die Methode summe an, die alle Elemente einer int-Liste aufsummiert und das Ergebnis zurückgibt:

```
public static int summe(int[] liste) {
   System.out.println("Summe von int[]");
   int summe = 0;
   for (int x : liste)
      summe += x;
   return summe;
}
```

Prinzip des Überladens II

Für die Aufsummierung einer double-Liste können wir eine weitere Methode anbieten, die den gleichen Methodennamen verwendet.

Wir ändern allerdings die Typen der Ein- und Ausgabeparameter zum vorhergehenden Beispiel:

```
public static double summe(double[] liste) {
   System.out.println("Summe von double[]");
   double summe = 0;
   for (double x : liste)
      summe += x;
   return summe;
}
```

Prinzip des Überladens III

Die Entscheidung, welche der beiden Methoden jetzt verwendet wird, liegt beim Aufrufer:

```
int[] intWerte = {1,2,3,4,5,6,7,8,9,10};
int summeIntWerte = summe(intWerte);
System.out.println(summeIntWerte);

double[] doubleWerte = {1,2,3,4,5,6,7,8,9,10};
double summeDoubleWerte = summe(doubleWerte);
System.out.println(summeDoubleWerte);
```

Es wird die passende Signatur ermittelt: Im ersten Beispiel passt der Eingabetyp int[] und im zweiten die double[].

Als Ausgabe erhalten wir:

```
Summe von int[]
55
Summe von double[]
55.0
```

Überladen ist ein sehr wichtiger und häufig verwendeter Mechanismus - deshalb lassen sich nicht nur Methoden überladen.

Überladen von Konstruktoren

Das Prinzip des Überladens lässt sich auch auf Konstruktoren übertragen. Nehmen wir wieder als Beispiel die Klasse Person und erweitern die Konstruktoren. Dann haben wir die Möglichkeit auch ein Objekt der Klasse Person zu erzeugen, wenn nur der Name, aber nicht das Alter bekannt sein sollte:

```
public class Person {
   private String name;
   private int alter;

   public Person() {
      setName("");
      setAlter(1);
   }

   public Person(String name) {
      setName(name);
      setAlter(1);
   }

   public Person(String name, int alter) {
      setName(name);
      setAlter(alter);
   }

   // get-und set-Methoden
   ...
}
```

Überladen von Konstruktoren II

Kleiner Hinweis an dieser Stelle: Genau dann, wenn kein Konstruktor angegeben sein sollte, fügt Java den sogenannten Default-Konstruktor hinzu, der keine Parameter erhält und keine Attribute setzt. Es kann trotzdem ein Exemplar dieser Klasse erzeugt werden.

Nun aber ein Beispiel zu der neuen Version der Klasse Person:

```
Person leer    = new Person();
Person herbert = new Person("Herbert", 30);
```

Die erste Zeile verwendet den passenden Konstruktor public Person() mit der parameterlosen Signatur und die zweite verwendet entsprechend den dritten Konstruktor.

Konstruktoren verketten

Bei genauerer Betrachtung der drei im letzten Abschnitt angegebenen Konstruktoren führen wir jeweils die gleichen zwei Funktionsaufrufe aus. Aus Vermeidung von Redundanz ist es sinnvoll, die Konstruktoren zu verketten, dazu gibt es beispielsweise die Möglichkeit über das Schlüsselwort this auf die klasseneigenen Konstruktoren zuzugreifen:

```
public Person() {
   this("", 1);
}

public Person(String name) {
   this(name, 1);
}

public Person(String name, int alter) {
   setName(name);
   setAlter(alter);
}
```

So rufen der erste und zweite Konstruktor jeweils den dritten auf.

Dabei ist darauf zu achten, dass einer der Konstruktoren die zwei Methoden wirklich aufrufen muss, da es ansonsten zu einer rekursiven Definition führt. Analog zum Schlüsselwort super gilt auch hier, dass this als erste Anweisung im Konstruktor stehen muss.

Der Copy-Konstruktor

Bei der Übergabe von Objekten werden diese nicht kopiert, es wird lediglich die Objektreferenz übergeben. Dürfen lokale Änderungen keinen Effekt auf das Objekt haben, müssen wir also immer daran denken, eine lokale Kopie des Objekts zu erstellen.

Elegant lässt sich das mit dem sogenannten Copy-Konstruktor lösen. Wir legen einfach bei der Implementierung einer Klasse, zusätzlich zu den vorhandenen Konstruktoren, einen weiteren Konstruktor an, der als Eingabeparameter einen Typ der Klasse selbst enthält, und kopieren den kompletten Inhalt.

Hier ein kurzes Beispiel, in dem ein Exemplar der Klasse Person kopiert wird:

```
public Person(Person original) {
   this(original.getName(), original.getAlter());
}
```

Wir können mit Hilfe des neuen Konstruktors Objektkopien erstellen:

```
Person herbert      = new Person("Herbert", 30);
Person herbertClone = new Person(herbert);
```

Da nur die Klasse selbst genau weiß, welche Abhängigkeiten zu anderen Klassen besteht und welche Objekte erzeugt wurden, kann im Gegensatz zur Methode clone(), die eine flache Kopie erstellt, eine tiefe Kopie des Objekts vorgenommen werden.

Statische Attribute und Methoden

Die Nutzung von Attributen und Methoden war bisher an die Existenz von konkreten Objekten gebunden:

```
Person herbert = new Person("Herbert", 30);
int alter = herbert.getAlter();
```

Um die Methode getAlter der Klasse Person verwenden zu können, müssen wir zuallererst ein Objekt der Klasse erzeugen.

Mit dem Schlüsselwort static können wir Attribute und Methoden, in von einem Objekt unabhängig verwendbare Klassenattribute und Klassenmethoden umwandeln. Jetzt ist die Verwendung nicht mehr an ein konkretes Objekt gebunden. Bei der Methode getAlter wäre das sicherlich nicht sinnvoll, denn die Ausgabe dieser Methode ist abhängig von einer konkreten Person.

Statische Methoden

Als wir mit den ersten Programmbeispielen in Java begonnen hatten, wurde jede Methode absichtlich mit static versehen (siehe Abschnitt [sub:Funktion-ohne-Rückgabewert]), da wir aus didaktischen Gründen die Einführung in die Objektorientierung auf einen späteren Zeitpunkt verlegt haben.

Das erste eigene Beispiel war die Methode gibAus:

```
public static void gibAus(int a) {
   System.out.println();
   System.out.println("*****************************************");
   System.out.println("*** Wert der Variable ist "+a);
   System.out.println("*****************************************");
   System.out.println();
}
```

Das Verhalten dieser Methode ist nicht abhängig von der Klasse in der sie steht, da diese nicht auf Attribute der Klasse zugreift. Die Abhängigkeit besteht hier in dem Eingabeparameter a.

Statische Methoden II

Schauen wir uns ein weiteres Beispiel an, in dem wir gleich mal den einzigen dreistelligen Operator von Java vorstellen wollen.

Die Syntax dafür sieht wie folgt aus:

<Variable> = <(Bedingung)> ? <Ausdruck1> : <Ausdruck2>;

Die Abarbeitung entspricht in etwa der folgenden Darstellung:

```
if (<Bedingung>)
   <Variable> = <Ausdruck1>;
else
   <Variable> = <Ausdruck2>;
```

Statische Methoden III

Jetzt wollen wir eine Methode max in der Klasse MeinMathe anbieten, die das Maximum zweier Eingaben a und b zurückliefern soll:

```
public class MeinMathe {
   public static int max(int a, int b) {
      return a<b ? b : a;
   }
}
```

Wir werden für die beiden folgenden Beispiele die Parameter a und b verwenden:

```
int a   = -3;
int b   = 22;
```

Da die Methode max statisch vorliegt, können wir sie über MeinMathe.max verwenden, ohne ein konkretes Objekt der Klasse MeinMathe erzeugen zu müssen:

```
int max = MeinMathe.max(-3, 22);
```

Sollten wir das Schlüsselwort static weglassen, so müssen wir wieder ein Objekt erzeugen und können über die Objektreferenz an die Methode gelangen:

```
MeinMathe mm = new MeinMathe();
int max = mm.max(a, b);
```

Jetzt ist auch klar, dass die Methode main, die von ausserhalb einmal beim Programmstart aufgerufen wird, ebenfalls static sein muss.

Statische Attribute

Analog verhält es sich bei dem Einsatz von statischen Attributen. Schauen wir uns ein Beispiel für die Verwendung von Klassenattributen an. Wir wollen dabei zählen, wie viele Exemplare einer Klasse Tasse erzeugt werden:

```
public class Tasse {
   private static int zaehler = 0;

   public Tasse() {
      zaehler++;
   }

   public static int getZaehler() {
      return zaehler;
   }
}
```

Testen wir unser Programm in dem wir den Parameter zaehler ausgeben, anschließend drei Tassen erzeugen und zaehler abschließend wieder ausgeben:

```
System.out.println("Tassen: " + Tasse.getZahler());

Tasse t1 = new Tasse();
Tasse t2 = new Tasse();
Tasse t3 = new Tasse();

System.out.println("Tassen: " + Tasse.getZahler());
```

Statische Attribute II

Wir greifen also direkt über die Klasse auf die Methode getZaehler zu. Als Ausgabe erhalten wir:

```
Tassen: 0
Tassen: 3
```

Unser Tassenbeispiel wird jetzt um eine nicht statische Methode erweitert, die in Abhängigkeit zu dem jeweiligen Objekt ebenfalls den Wert zaehler ausgeben soll:

```
public int getZaehler() {
   return zaehler;
}
```

Interessanterweise haben wir die gleiche Signatur verwendet und die Methoden nur durch static unterschieden. Aber durch den Kontext des Aufrufers ist immer klar, um welche der beiden Methoden es sich handelt.

Jetzt können wir die drei erzeugten Objekte ebenfalls nach dem Wert von zaehler befragen:

```
Tasse t1 = new Tasse();
Tasse t2 = new Tasse();
Tasse t3 = new Tasse();

System.out.println("Tassen: "+t1.getZaehler());
System.out.println("Tassen: "+t2.getZaehler());
System.out.println("Tassen: "+t3.getZaehler());
```

Statische Attribute III

Wir sehen, dass der Aufruf an die konkrete Objektreferenz geknüpft wurde und erhalten verständlicherweise:

```
Tassen: 3
Tassen: 3
Tassen: 3
```

Jetzt stellt sich noch die Frage, wann wir ein Attribut oder eine Methode mit dem Schlüsselwort static versehen sollen und wann nicht.

In der Regel lässt sich das wie folgt entscheiden: Sollte ein Attribut oder eine Funktion unabhängig von den jeweiligen Objekten sein, z. B. die Funktion max aus der Klasse MeinMathe, dann sollten diese mit static versehen werden. Sind allerdings Attribut oder Methode abhängig von einem konkreten Objekt, so wie z. B. die Funktion getAlter in der Klasse Person, dann darf kein static davor stehen.

Konstruktoren ganz privat

In vielen Beispielen wurde der durch die Verwendung von Konstruktoren entstehende Komfort bei der Objekterstellung aufgezeigt. Bei der Einführung von Konstruktoren haben wir als Modifizierer für Konstruktoren public verwendet. Das ist sicherlich sinnvoll, da wir ansonsten nicht auf die Konstruktoren zugreifen können, um Objekte zu erstellen.

Aus einer abstrakteren Sicht auf den Entwurf von Software kann es durchaus sinnvoll sein, die Erzeugung von Klassen zu verbieten oder gegebenenfalls zu reglementieren.

Klassenexemplare nicht erlaubt

Verbieten könnten wir beispielsweise den Zugriff auf Konstruktoren einer Klasse, die nur statische Methoden aufweist. Wir haben im vorhergehenden Abschnitt die Klasse MeinMathe kennengelernt.

Nehmen wir an, dass weitere statische Methoden für die Ermittlung eines Minimums oder die Berechnung von sin und cos vorhanden sind:

```
int    min  = MeinMathe.min(-3, 22);
double sinX = MeinMathe.sin(1.25);
double cosX = MeinMathe.cos(2.005);
```

Jetzt können wir innerhalb der Klasse MeinMathe durch einen privaten Konstruktor verhindern, dass ein Objekt dieser Klasse erzeugt wird:

```
private MeinMathe() {}
```

Mit der Definition eines eigenen leeren Konstruktors haben wir die Erstellung eines Defaultkonstruktors unterbunden. Jetzt ist es nicht mehr möglich, ein Exemplar dieser Klasse zu erzeugen:

```
MeinMathe mm = new MeinMathe();
```

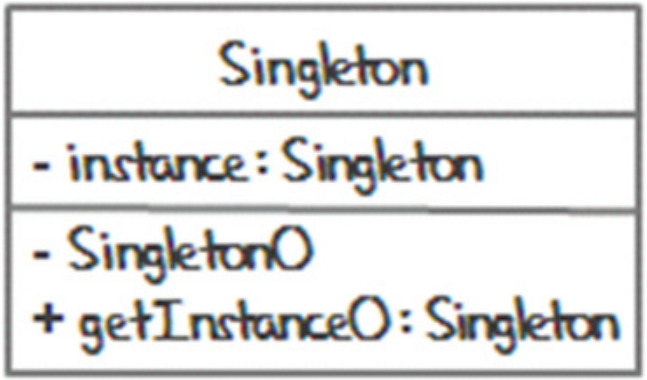
Wir erhalten einen Kompilierfehler.

Geschützte Konstruktoren sind nicht nur für Klassen mit statischen Methoden interessant, wie uns der folgende Abschnitt zeigen wird.

Ein Einzelstück ist erlaubt

Angenommen, dass wir die Aufgabe erhalten haben, einen Mechanismus zu realisieren, der nur die Erzeugung eines einzelnen Exemplars einer Klasse erlaubt. In der Praxis werden oft globale Klassen, die beispielsweise Daten protokollieren oder Konfigurationen enthalten als Einzelstücke (singleton) realisiert. Für diese Aufgaben hat sich ein Entwurfsmuster etabliert.

Über die Methode getInstance wird beim ersten Aufruf das Einzelstück erstellt und bei weiteren Aufrufen die Referenz zu dem Einzelstück zurückgeliefert.



Hier sehen wir eine kleine Erweiterung zu der bereits beschriebenen Variante zur Darstellung von Klassen in UML. Die Zeichen vor den Attributen und Methoden symbolisieren die Modifizierer public (+), private (-) und protected (#). Auf den dritten Modifizierer werden wir im nächsten Kapitel noch genauer zu sprechen kommen.

Singleton Implementierung

Hier sehen wir die Implementierung der Singleton-Klasse:

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton() {};

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

   public Object clone() throws CloneNotSupportedException {
      throw new CloneNotSupportedException("Clonen nicht erlaubt");
   }
}
```

Wir müssen noch darauf achten, dass wir den clone-Mechanismus ausschalten. Dafür wird die Methode einfach überschrieben und liefert einen Fehler, falls trotzdem versucht wird, das Einzelstück zu clonen.

Singleton Implementierung

Hier sehen wir die Implementierung der Singleton-Klasse:

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton() {};

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

   public Object clone() throws CloneNotSupportedException {
      throw new CloneNotSupportedException("Clonen nicht erlaubt");
   }
}
```

Wir müssen noch darauf achten, dass wir den clone-Mechanismus ausschalten. Dafür wird die Methode einfach überschrieben und liefert einen Fehler, falls trotzdem versucht wird, das Einzelstück zu clonen.

Garbage Collector

Bisher haben wir andauernd Speicher für unsere Variablen mit new reserviert, aber wann wird dieser Speicher wieder freigegeben?

Java besitzt mit dem Garbage Collector eine „Müllabfuhr", die sich automatisch um die Freigabe des Speichers kümmert. Das ist gegenüber anderen Programmiersprachen ein großer Komfort. Dieser Komfort ist allerdings nicht kostenlos, denn Java entscheidet eigenständig über den Zeitpunkt, den Garbage Collector zu starten. Es könnte also auch dann geschehen, wenn das Programm gerade eine rechenintensive Aufgabe zu erfüllen hat.

Es gibt allerdings die Möglichkeit, mit System.gc() den Start des Garbage Collectors zu empfehlen -- daran halten muss sich die virtuelle Maschine allerdings nicht.

Um es nochmal deutlich zu machen: In Java braucht sich der Programmierer nicht um die Speicherfreigabe zu kümmern.

Finale Klassen

In der Softwareplanung kann es vorkommen, dass Klassen beschrieben werden, die nicht weiter vererbt werden dürfen. Um die Klassen zu Schützen und die Vererbung zu unterbinden, fügen wir das Schlüsselwort final hinzu:

```
public final class FinaleKlasse {
}
```

Wenn wir jetzt versuchen sollten, von der Klasse FinaleKlasse abzuleiten, würde der Kompiler eine Fehlermeldung liefern.

Standardwerte bei der Objekterzeugung

Im Unterschied zu einfachen primitiven Datentypen wird bei der Objekterzeugung den primitiven Datentypen immer ein Defaultwert vergeben. So werden alle Zahlentypen mit 0, boolean mit false und Referenztypen mit null initialisiert. Wir haben uns in diesem Beispiel die get- und set-Methoden gespart, um das Programm kompakt zu halten:

```
public class InitTest {
   private byte   b;
   private short  s;
   private int    i;
   private long   l;
   private float  f;
   private double d;
   private String string;

   public InitTest() {
   }

   public static void main(String[] args) {
      InitTest iT = new InitTest();
      System.out.println("byte:\t "+iT.b);
      System.out.println("short:\t "+iT.s);
      System.out.println("int:\t "+iT.i);
      System.out.println("long:\t "+iT.l);
      System.out.println("float:\t "+iT.f);
      System.out.println("double:\t "+iT.d);
      System.out.println("String:\t "+iT.string);
   }
}
```

Standardwerte bei der Objekterzeugung II

Wir erhalten jetzt die erwarteten Defaultwerte:

```
byte:    0
short:   0
int:     0
long:    0
float:   0.0
double:  0.0
String:  null
```

Trotzdem hier noch einmal der Hinweis: Es gehört zum guten Programmierstil die Initialisierung der vorhandenen Attribute, beispielsweise über einen Konstruktor eigenständig vorzunehmen.

Standardwerte bei der Objekterzeugung III

Es gibt auch noch die Möglichkeit einen Initialisierungsblock zu verwenden, der bei dem Erzeugen eines Objekts noch vor dem Konstruktor aufgerufen wird:

```
public class InitBlock {
   private int    i;
   private double d;
    private String string;

    {
       System.out.println("Initialisierungsblock wird aufgerufen");
       i          = 1;
       d          = 1;
       string     = "leer";
    }

    public InitBlock() {
       System.out.println("Konstruktor wird aufgerufen");
    }

    public static void main(String[] args) {
       InitTest iT = new InitTest();
    }
}
```

Zunächst wird der Initialisierungsblock aufgerufen, dann der Konstruktor.

Verwendung von Bibliotheken

Übersicht zum Vorlesungsinhalt

zeitliche Abfolge und Inhalte können variieren

Verwendung von Bibliotheken

- Standardbibliotheken
- Klasse Math
- Zufallszahlen, Verteilungen
- Lineare Algebra – JAMA
- Eigene Bibliotheken erstellen
- Java API
- Javadoc

Pakete einbinden und verwenden

Um Methoden und Klassen einer Bibliothek in einer Klasse bekannt zu machen, können wir diese gleich zu Beginn explizit einbinden. Hier die Syntax, um eine konkrete Klasse Y einzubinden, die in einem dem System bekannten Ordner x liegt:

```
import x.Y;
```

Wenn wir alle Klassen, die in einem Ordner x liegen einbinden möchten, können wir einen * angeben:

```
import x.*;
```

Schauen wir uns dazu nocheinmal die für uns in diesem Kontext relevanten Zeilen des Einlesebeispiels an:

```
 ...
import java.util.Scanner;

public class DateiAuslesen {
   public static void main(String[] args) {
      ...
      Scanner scanner = new Scanner(new File(dateiName));
      ...
   }
}
```

Pakete einbinden und verwenden II

Bevor der Inhalt der Klasse DateiAuslesen beschrieben wird, machen wir mit dem Schlüsselwort import die Klasse Scanner für unsere Klasse bekannt, die in einem Verzeichnis util liegt, das widerum in einem dem System bekannten Verzeichnis java zu finden ist. Wir haben die Klasse Scanner eingebunden und können diese verwenden.

Alternativ können wir eine Klasse Y aus einem Verzeichnis x direkt ansprechen, ohne das Schlüsselwort import zu verwenden:

```
 ...
public class DateiAuslesen {
   public static void main(String[] args) {
      ...
      java.util.Scanner scanner =
         new java.util.Scanner(new File(dateiName));
      ...
   }
}
```

Wenn wir diese implizite Variante einsetzen, müssen wir allerdings an jeder Stelle den kompletten Pfad angeben.

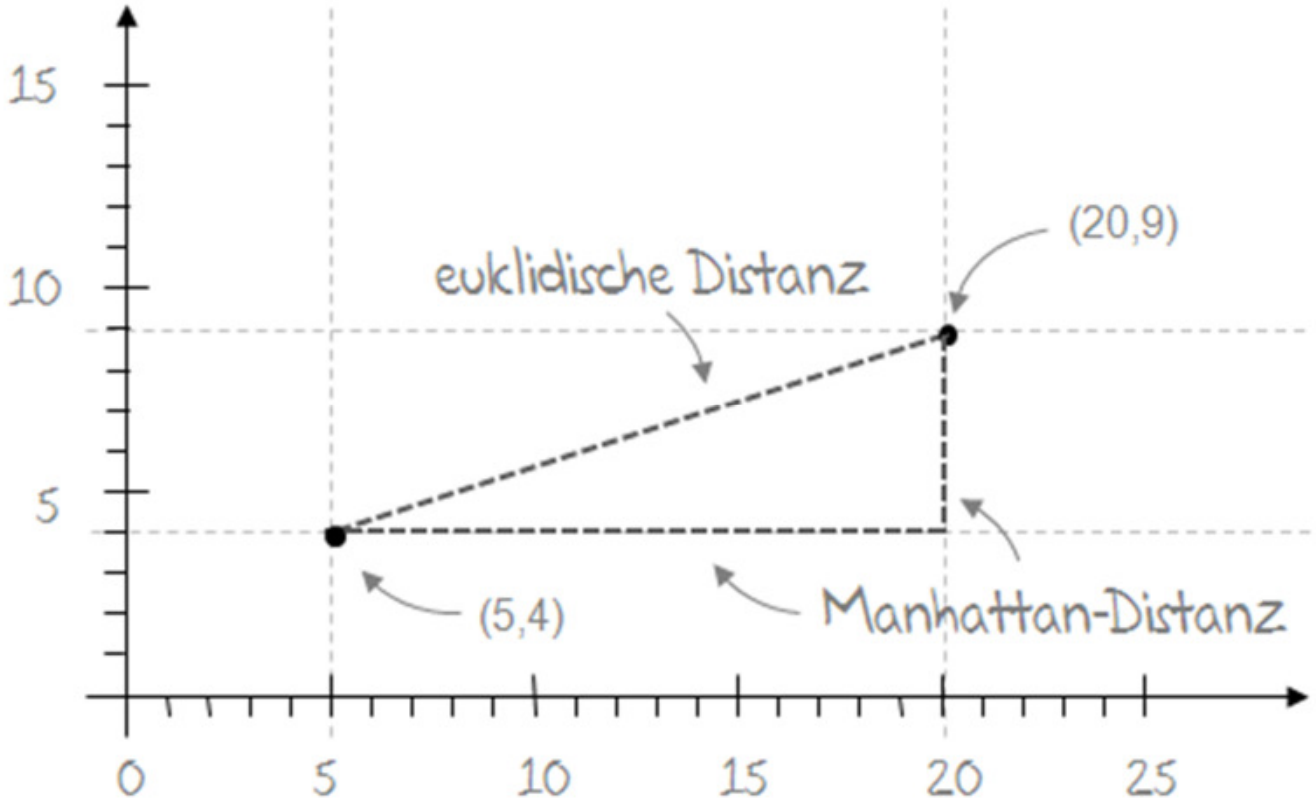
Methoden der Klasse Math

Die Methoden der Klasse Math werden häufig benötigt. So bietet die Klasse neben einigen trigonometrischen Funktionen, wie sin und cos, auch Funktionen an, wie min, max und log. Es gibt noch sehr viele weitere. Auch die Konstanten PI und E lassen sich dort entdecken.

Einfache Abstände berechnen

Für die folgenden Beispiele wollen wir uns die Klasse Point aus dem Paket java.awt ausleihen. Ein Exemplar dieser Klasse Point repräsentiert einen zweidimenionalen Punkt bestehend aus einem x- und einem y-Wert.

Wir wollen im folgenden Beispiel den euklidischen Abstand zwischen zwei gegebenen Punkten berechnen:



Einfache Abstände berechnen II

Da die Methoden in der Klasse Math statisch definiert sind, können alle direkt verwendet werden, wie in dem folgenden Beispiel gezeigt:

```
import java.awt.Point;

public class MathematikTest {
   public static double eukDistanz(Point p1, Point p2) {
      // wir quadrieren die Differenz beider x-Werte
      double diffX = Math.pow(p2.x-p1.x, 2);
      double diffY = Math.pow(p2.y-p1.y, 2); // ... und beider y-Werte

      // schließlich wird die Quadratwurzel aus der Summe
      // der beiden Ergebnisse gezogen
      double dist = Math.sqrt(diffX + diffY);
      return dist; // ... und zurück geliefert
   }

   public static void main(String[] args) {
      Point p1 = new Point( 5, 4);
      Point p2 = new Point(20, 9);

      System.out.println("Euklidische Distanz: "+eukDistanz(p1, p2));
   }
}
```

Euklidische Distanz und Manhattan-Distanz

Dieses Beispiel berechnet über die Methode eukDistanz den euklidischer Abstand zwischen zwei Punkten p1 und p2:

$$eukDistanz(p_1, p_2) = \sqrt{(p_2.x - p_1.x)^2 + (p_2.y - p_1.y)^2}$$

Als zweite Abstandsmetrik wollen wir kurz den Manhattan-Abstand vorstellen, der die Distanz zwischen zwei Punkten über die Summe der absoluten Differenzen der Einzelkoordinaten angibt:

$$manDistanz(p_1, p_2) = |p_2.x - p_1.x| + |p_2.y - p_1.y|$$

Für die Umsetzung können wir die Methode abs einsetzen:

```
public static double manDistanz(Point p1, Point p2) {
   return Math.abs(p2.x-p1.x) + Math.abs(p2.y-p1.y);
}
```

Die Bezeichnung Manhattan-Abstand ist auf die Gebäude- und Straßenanordnung im gleichnamigen Stadtbezirk von New York zurückzuführen.

Fläche und Umfang eines Kreises berechnen

Aus dem Mathematikunterricht wissen wir, dass sich die Fläche A eines Kreises mit dem Radius r wie folgt berechnet:

$$A_{Kreis} = \pi r^2$$

Wir können dafür die in der Klasse Math definierten Konstante PI und die Methode pow verwenden:

```
public static double flaecheKreis(double r) {
   return Math.PI * Math.pow(r, 2);
}
```

Der Umfang U eines Kreises mit dem Radius r ergibt sich aus:

$$U_{Kreis} = 2\pi r$$

Auch diese Umsetzung ist jetzt sehr einfach:

```
public static double umfangKreis(double r) {
   return 2 * Math.PI * r;
}
```

Zufallszahlen mit Math

In der Mathebibliothek gibt es mit random auch eine Methode zur Erzeugung von Zufallszahlen, die haben wir bereits in verschiedenen Projekten kennengelernt.

Wie wollen beispielsweise eine gleichverteilte, ganzzahlige Zufallszahl aus dem Intervall [0,9] erzeugen:

```
int zuffi = (int)(Math.random()*10);
```

Eine Verteilung wird als gleichverteilt bezeichnet, wenn jeder Wert der Verteilung die gleiche Wahscheinlichkeit besitzt, gezogen zu werden. Oder allgemeiner eine gleichverteilte, ganzzahlige Zufallszahl aus dem Intervall [n, m] erzeugen, mit n<m:

```
int zuffi = (int)(Math.random()*(m-n)+n);
```

Für den Fall, dass eine reelle Zufallszahl aus dem Intervall [n, m] mit n<m erzeugt werden soll, können wir die Typumwandlung einfach weg lassen:

```
double zuffd = Math.random()*(m-n)+n;
```

Die die Arbeit mit Zufallszahlen aber einen großen Anwendungsbereich hat, bietet die Klasse Random noch mehr Funktionalität an.

Zufallszahlen mit Random

Es gibt verschiedene Möglichkeiten, Zufallszahlen in Java zu erzeugen. Mit Math.random steht uns bereits eine Methode zur Verfügung. Jetzt wollen wir die im vorhergehenden Abschnitt vorgestellten Beispiele mit der Methoden der Klasse Random umsetzen. Dazu müssen wir zunächst ein Exemplar der Klasse erzeugen:

```
Random gen = new Random();

// gleichverteilt, ganzzahlig aus [0, 9]
int zuffi    = gen.nextInt(10);

// gleichverteilt, ganzzahlig aus [n, m] mit n<m
zuffi        = gen.nextInt(m-n+1)+n;

// gleichverteilt, reell aus [n, m] mit n<m
double zuffd = gen.nextDouble()*(m-n+1)+n;
```

Mit der Methode nextInt(int n) wird eine ganzzahlige Zufallszahl aus dem Bereich [0,n), also 0 inklusive und n exklusive, erzeugt.

Die Methoden nextFloat und nextDouble liefern Zufallszahlen aus dem Bereich [0,n) mit der entsprechenden Bitanzahl. Es gibt noch weitere Methoden in der Klasse Random.

Simulation: Lotto 6 aus 49

Als kleine Anwendung werden wir eine Lotterieziehung simulieren. Dazu werden wir 6 Zahlen aus dem Bereich [1,49] ohne zurücklegen zufällig ziehen und die Liste der gezogenen Zahlen zurückliefern:

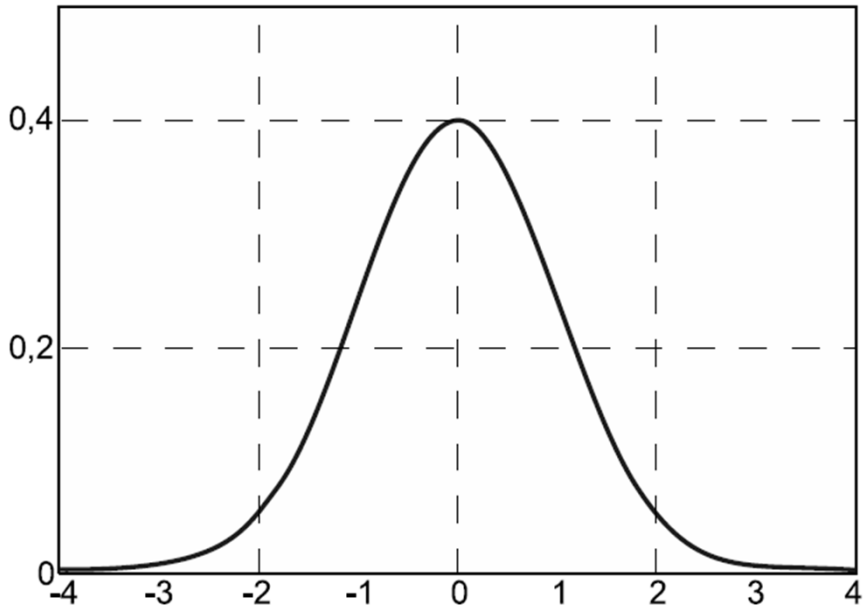
```
public static int[] lotterie() {
   Random rg = new Random();
   int[] zuf = new int[6];
   int wert, i=0;

   aussen:
   while (i<6) {
      wert = rg.nextInt(49)+1;

      // wert in zuf[] schon vorhanden?
      for (int j=0; j<i; j++)
         if (zuf[j] == wert)
            continue aussen;

      zuf[i] = wert;
      i++;
   }
   return zuf;
}
```

So oder so ähnlich könnten die Lottozahlen der nächsten Ziehung aussehen:

```
Lotto (6 aus 49): 16   13   42   40   44   45
```

Normalverteilte Zufallszahlen

Bisher hatten wir für die gegebenen Zahlenmengen gefordert, dass jede Zahl die gleiche Wahrscheinlichkeit besitzt, gezogen zu werden. Eine weitere sehr wichtige Verteilung ist die Normalverteilung oder Gauß-Verteilung, bei der sich die gezogenen Zufallszahlen in der Mitte der Verteilung konzentrieren und mit größerem Abstand zur Mitte immer seltener auftreten:



Funktionskurve der Standardnormalverteilung mit dem Mittelwert 0 und der Standardabweichung 1. Knapp 70% der Zufallszahlen liegen dann zwischen -1 und 1, 95% zwischen -2 und 2 und fast 100% zwischen -3 und 3.

Normalverteilte Zufallszahlen I

Die Dichtefunktion der Standardnormalverteilung mit dem Mittelwert 0 und einer Standardabweichung von 1 wird durch folgende Formel beschrieben:

$$f(x) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{1}{2}x^2}$$

Wir können beispielsweise mit einem kleinen Würfel-Experiment eine Verteilung erzeugen, die der Normalverteilung sehr ähnlich ist. Gegeben seien dabei zwei Würfel, deren Zahlen gleichverteilt erscheinen. Wir würfeln beide und addieren die Werte. Möglich sind die elf unterschiedlichen Ergebnisse 2 bis 12.

Hier ein kleiner Programmabschnitt, der die Ergebnisse der Einzelexperimente in einer Liste verteilung speichert und zurückgibt:

```
public static int[] wuerfelExperiment(int n) {
   Random rg = new Random();
   int[] verteilung = new int[11];

   for (int i=0; i<n; i++)
      verteilung[rg.nextInt(6) + rg.nextInt(6)]++;

   return verteilung;
}
```

Normalverteilte Zufallszahlen II

Wenn wir diese Methode nur ein paar mal anwenden (Zeile 1: 10), wirkt die Verteilung zufällig. Bei mehreren Durchläufen (Zeile 2: 100, Zeile 3: 1000) ähnelt diese aber der Normalverteilung mit einem Erwartungswert (so wird der Mittelwert dieser Verteilung genannt) von 7:

```
10:    0    0    3    1    2    2    0    1    0    0    1
100:   1    8    7   10   10   12   18   14    7   10    3
1000: 20   62   93  112  125  171  150  118   70   49   30
```

Für unser Experiment genügte es, Zufallszahlen zwischen 0 und 5 zu erzeugen und diese in eine elf-elementige Liste zu speichern.

Deterministische Zufallszahlen

Die Erzeugung von Zufallszahlen in einem Computer basiert auf Methoden, die sogenannte Pseudozufallszahlen erzeugen. Dabei ist der Startpunkt einer Zufallszahlenfolge entscheidend. Die Klasse Random bietet beispielsweise zwei Konstruktoren an. Die parameterlose Variante haben wir bereits gesehen. Bei dieser initialisiert sich die Folge in Abhängigkeit zu der aktuellen Systemzeit und erzeugt bei jedem Start neue Folgen von Zufallszahlen.

Um beispielsweise komplexe Programme zu testen, in denen Zufallszahlen eine Rolle spielen, können zufällige Ergebnisfolgen das Debuggen sehr erschweren. Nachdem ein potentieller Fehler identifiziert und korrigiert wurde, sollte eigentlich die gleiche zufällige Ergebnisfolge getestet werden, um sicher gehen zu können, dass der Fehler kein zweites Mal auftritt.

Für solche Fälle gibt es einen parametrisierten Konstruktor mit einem long, bei dem die Folge der Zufallszahlen nach dem Start immer gleich (also deterministisch) ist. Wir können beispielsweise einen long mit dem Wert 0 immer als Startwert, dem so genannten seed-Wert, nehmen und erhalten anschließend immer dieselben Zufallszahlen:

```
long initwert = 0;
Random rGen   = new Random(initwert);
```

Das bedeutet sogar, dass wir den Computer zwischendurch ausschalten können.

Zufallszahlen sicher und schnell

Ein interessantes Phänomen lässt sich beobachten, wenn wir es darauf anlegen, die größtmögliche Zufallszahl mit der Methode nextDouble aus der Klasse Random zu erzeugen. Wir haben bereits mehrfach erläutert, dass der Wertebereich bei [0,1) liegt, also die 1 nicht erreicht werden kann. Das ist auch soweit richtig, allerdings kann in einem Kontext dieses Ergebnis sehr wohl auftreten.

Angenommen wir haben einen Wert sehr nahe bei 1, z. B. den Wert d in dem folgenden Beispiel:

```
double d = 0.9999999999999999;
System.out.println(d+", "+(d+1));
```

Wenn wir diesen darauf überprüfen, ob der Wert eine 1 darstellt, werden wir immer ein false erhalten. Addieren wir allerdings zu diesem Wert eine 1 erhalten wir eine 2:

```
0.9999999999999999, 2.0
```

Die Methode Math.random basiert intern auf nextDouble und wir können Pech haben, dass der ermittelte Wert so Nahe bei der 1 liegt, dass wir ein unerwartetes Ergebnis erhalten. Sollten wir dann den folgenden, bekannten Weg einschlagen, um ein Zufallszahlintervall zu definieren, kann das fehlschlagen:

```
int zuffi = (int)(Math.random()*n);
```

Wir sollten für diese Fälle also immer die Methode nextInt verwenden.

Zufallszahlen sicher und schnell II

Ein zweiter wichtiger Punkt beim Umgang mit Zufallszahlen ist sicherlich die Geschwindigkeit. Die Methode random in Math basiert, wie bereits gesagt, auf nextDouble in Random. Daher wird der indirekte Aufruf sicherlich noch etwas Zeit kosten. Mit der Klasse ThreadLocalRandom wird seit Java 1.7 eine Randomklasse angeboten, bei der lediglich ein Exemplar der Klasse verwendet wird (Singleton-Muster):

Folgendes Beispiel führt einen kleinen Performanzvergleich zwischen den drei genannten Varianten durch, bei dem jeweils 10.000.000 Zufallszahlen erzeugt werden:

```
long startZeit = System.currentTimeMillis();
for (int i=0; i<10000000; i++) Math.random();
long stopZeit  = System.currentTimeMillis();
System.out.println("Dauer (Math.random)      "+(stopZeit-startZeit)+" ms");

Random r1 = new Random();
startZeit = System.currentTimeMillis();
for (int i=0; i<10000000; i++) r1.nextDouble();
stopZeit  = System.currentTimeMillis();
System.out.println("Dauer (Random)           "+(stopZeit-startZeit)+" ms");

Random r2 = ThreadLocalRandom.current();
startZeit = System.currentTimeMillis();
for (int i=0; i<10000000; i++) r2.nextDouble();
stopZeit  = System.currentTimeMillis();
System.out.println("Dauer (ThreadLocalRandom)"+(stopZeit-startZeit)+" ms");
```

Zufallszahlen sicher und schnell III

Das ergab die folgende Ausgabe:

```
Dauer (Math.random)       3984 ms
Dauer (Random)            3610 ms
Dauer (ThreadLocalRandom)1875 ms
```

Wir sehen den großen Zeitunterschied. Demzufolge ist der Einsatz der Klasse ThreadLocalRandom für die Erzeugung von Zufallszahlen zu bevorzugen.

Live-Coding-Session

Installation der JAMA-Bibliothek

Um JAMA zu installieren, gehen wir zunächst zu dem aufgeführten Link:
http://math.nist.gov/javanumerics/jama/

Wir speichern nun das Zip-Archiv vom Source (zip archive, 105Kb) z.B. in den Ordner "c:\Java\". Jetzt ist das Problem, dass die bereits erstellten .class-Dateien nicht zu unserem System passen. Desshalb müssen die sechs ".class" Dateien im Ordner Jama löschen und zusätzlich das Class-File im Ordner util. Jetzt ist es quasi clean.

Nun gehe in den Ordner "c:\Java" und gebe folgendes ein:

```
C:\Java>javac Jama/Matrix.java
  Note: Jama\Matrix.java uses unchecked or unsafe operations.
  Note: Recompile with -Xlint:unchecked for details.J
```

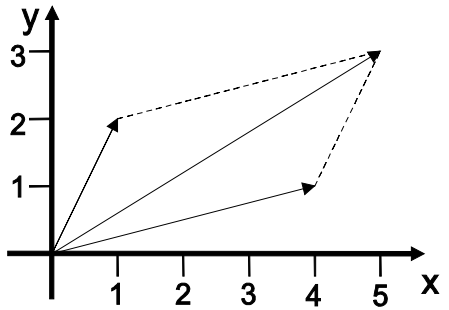
Jetzt wurde das Package erfolgreich compilert.

Lineare Algebra I

Es gibt viele nützliche Bibliotheken, die wir verwenden können. JAMA ist beispielsweise eine oft verwendete Bibliothek für Methoden der Linearen Algebra.

Hier ein Beispiel zur Vektoraddition:

```
import Jama.*;

public class JAMATest{
   public static void main(String[] args){
      double[][] vector1 = {{1},{2}};
      double[][] vector2 = {{4},{1}};

      Matrix v1 = new Matrix(vector1);
      Matrix v2 = new Matrix(vector2);

      Matrix x  = v1.plus(v2);

      System.out.println("Matrix x: ");
      x.print(1, 2);
   }
}
```



Lineare Algebra II

Nun wollen wir die Determinante einer Matrix berechnen:

```
import Jama.*;

public class JAMATest{
   public static void main(String[] args){
      double[][] array = {{-2,1},{0,4}};
      Matrix a = new Matrix(array);
      double d = a.det();

      System.out.println("Matrix a: ");
      a.print(1, 2);

      System.out.println("Determinante: " + d);   // det(a) = (-2)*4 - 1*0 = -8
   }
}
```

Musikdateien abspielen mit AudioClip

Eine einfache Möglichkeit eine Musikdatei abzuspielen, bietet die Klasse AudioClip aus dem Paket applet an. In dem folgenden Beispiel lesen wir die Musikdatei bach.wav ein und spielen diese ab:

```
import java.applet.Applet;
import java.applet.AudioClip;
import java.net.URL;

public class MusikAbspielen {
   public static void main(String[] args) {
      AudioClip sound = null;
      try {
         sound = Applet.newAudioClip(
            new URL("file:///c://bach.wav"));
      }
      catch(Exception ex) {
         ex.printStackTrace();
      }
      sound.play();

      ...
   }
}
```

Dazu haben wir zunächst die Klasse URL aus dem Paket net verwendet, um den Ort einer lokal vorliegenden Musikdatei netzwerkspezifisch anzugeben. Anschließend wurde die Datei mit Hilfe der Klasse Applet aus dem Paket applet geladen. Über die Methode play wird die Datei einmal abgespielt und alternativ in einer Schleife über die Methode loop.

Eigene Bibliothek erstellen

Die Erzeugung eines eigenen Pakets unter Java ist sehr einfach. Wichtig ist das Zusammenspiel aus Klassennamen und Verzeichnisstruktur. Angenommen wir wollen eine Klasse MeinMax in einem Paket meinMathe anbieten. Dann legen wir ein Verzeichnis meinmathe an und speichern dort z. B. die folgende Klasse:

```
package meinmathe;

public class MeinMax{
   public static int maxi(int a, int b){
      if (a<b) return b;
      return a;
   }
}
```

Durch das Schlüsselwort package signalisieren wir, dass es sich um eine Klasse des Pakets meinmathe handelt. Laut Konvention verwenden wir ausschließlich Kleinbuchstaben. Firmen verwenden beispielsweise oft ihre umgedrehte Internetdomain: com.example.mypackage.

Unsere kleine Matheklasse bietet eine bescheidene maxi-Funktion.

Eigene Bibliothek erstellen II

Nachdem wir diese Klasse kompiliert haben, können wir außerhalb des Ordners eine neue Klasse schreiben, die dieses Paket testweise verwendet. Dabei ist darauf zu achten, dass der Ordner des neuen Paketes entweder im gleichen Ordner wie die Klasse liegt, die das Paket verwendet, oder dieser Ordner im CLASSPATH aufgelistet ist.

Hier unsere Testklasse:

```
import meinmathe.MeinMax;

public class MatheTester{
   public static void main(String[] args){
      System.out.println("Ergebnis = "+MeinMax.maxi(3,9));
   }
}
```

Wir erhalten nach der Ausführung folgende Ausgabe:

```
C:\JavaCode>java MatheTester
Ergebnis = 9
```

Das kleine Beispiel hat uns gezeigt, wie es mit wenigen Handgriffen möglich ist, unsere Methoden zu Paketen zusammenzufassen. Die Verzeichnisstruktur kann natürlich noch beliebig verschachtelt werden.

JavaDoc

Eine ausreichende und nachvollziehbare Kommentierung von Programmen ist ein wichtiger und absolut notwendiger Bestandteil bei der Entwicklung von Software. Wir haben mit dem einzeiligen und dem mehrzeiligen Kommentar bereits zwei wichtige Kommentierungsvarianten kennengelernt:

```
// Einzeiliger Kommentar

/* Kommentar geht über mehrere
   Zeilen */
```

Mit JavaDoc gibt es eine weitere Variante, die zusätzlich die Möglichkeit bietet, die kommentierten Klassen und beschriebenen Methoden in ein HTML-Format zu überführen und damit eine Dokumentation der aktuellen Programme vorzunehmen. Wir werden im Folgenden nur eine kurze Einführung geben.

Ein solcher Kommentar beginnt mit /** und endet ebenfalls mit */:

```
/**
 * Ich bin ein besonderer Kommentar
 */
```

Die führenden Sternchen vor jeder Zeile sind in neueren JavaDoc-Versionen nicht mehr notwendig.

Spezielle Elemente eines Kommentars

```
/**
 * Berechnet den Manhattan-Abstand zwischen zwei Punkten. <p>
 * Die Bezeichnung Manhattan-Abstand ist auf die Gebäude-
 * und Straßenanordnung im gleichnamigen Stadtbezirk von
 * New York zurückzuführen.
 *
 * [@author  ...]
 * [@version ...]
 *
 * @param p1 erster 2d-Punkt
 * @param p2 zweiter 2d-Punkt
 *
 * @return Manhattan-Abstand
 * [@throws ...]
 * [@see    ...]
 */
public static double manDistanz(Point p1, Point p2) {
   ...
```

In der Konsole navigieren wir in den Ordner, dessen Klassen wir dokumentieren wollen. Für unser Beispiel werden wir die kommentierte Klasse MathematikTest verwenden:

```
C:\JavaCode\kapitel8>javadoc MathematikTest.java
```