Java Cheat Sheet

Eine ganz, ganz kurze Einführung in Java
(ohne Anspruch auf Vollständigkeit)
für Leute, die schon programmieren können.

Version: 0.6
Datum: 8.12.2007
Autor: Sascha Leib

1. Hello World

Das folgende Beispiel zeigt das Grundgrüst für ein einfaches Java-Programm:

```
class HelloWorld
{
      public static void main (String args[])
      {
            System.out.println("Hallo Welt!");
            System.exit(0);
      }
}
```

Zur Erklärung:

- Der Name der Datei (plus Dateiendung „`.java`“) muss mit dem Namen der Klasse (hier „`HelloWorld`“) übereinstimmen - in diesem Fall also „`HelloWorld.java`“.
- Es wird zuerst die Funktion „`main`“ aufgerufen. Diese muss hier statisch deklariert sein („`static`“), da keine Klasseninstanz erzeugt wird, und sie muss öffentlich sein („`public`“), damit sie ‚von außen‘, also in diesem Fall beim Programmstart, aufgerufen werden kann.
- Der Rückgabewert des Programmes wird in Java nicht als Funktions-Rückgabe spezifiziert, sondern am Ende als Parameter von „`System.exit`“ (hier ‚`0`‘), daher hat „`main`“ einen leeren Rückgabetyp („`void`“).
- Die Funtion „`main`“ erhält als Parameter ein Array (gekennzeichnet durch die eckigen Klammern) von Strings. Dies enthält ggf. die Kommandozeilenparameter, die beim Programmstart mitgegeben werden.
- Die vordefinierten Funktionsaufrufe sind in einer Klassenbibliothek organisiert. Die Klasse „`System`“ enthält z.B. zahlreiche systemnahe Funktionen und Unterklassen, die sich mit Laufzeitsteuerung (z.B. hier: „`exit`“) oder Ein- und Ausgabe (z.B. „`out`“) befassen.

2. Kompilieren und Starten

Java-Sourcedateien haben die Dateiendung „`.java`“ und der eigentliche Dateiname sollte mit dem Klassennamen übereinstimmen. Das obige Programm sollte man also unter dem Namen „`HelloWorld.java`“ abspeichern.

Zum Kompilieren auf der Kommandozeile gibt man dann einfach ein:

```
> javac HelloWorld.java
```

Bei Erfolg erzeugt `javac` dann eine neue Datei mit der Endung „`.class`“, hier also entsprechend „`HelloWorld.class`“. Diese kann dann mit dem folgenden Befehl gestartet werden:

```
> java HelloWorld
```

Wichtig ist hier, dass `java` nicht (direkt) nach der Datei sucht, sondern nach dem Klassennamen – der natürlich schon etwas mit dem Dateinamen zu tun hat; Jedenfalls hier bitte keine Dateiendung angeben, sonst erscheint eine Fehlermeldung!

3. Datentypen

Java kennt die folgenden Grunddatentypen:

Datentyp	Standardwert	Beschreibung	Suffix
boolean	`false`	Wahrheitswert (`true` oder `false`)	
char	`'\u0000'`	Ein einzelnes Zeichen (`\u0000` bis `\uFFFF`*)	
byte	`0`	Ganzzahl 8-Bit mit Vorzeichen (-128 bis 127)	
short	`0`	Ganzzahl 16-Bit mit Vorzeichen (-32768 bis 32767)	
int	`0`	Ganzzahl 32-Bit mit Vorzeichen	
long	`0L`	Ganzzahl 64-Bit mit Vorzeichen	`L`
float	`0.0F`	32-Bit Fließkommazahl	`F`
double	`0.0D`	64-Bit Fließkommazahl	`D`

4.1 Ganzzahltypen

Für Variablen der Typen `byte`, `short`, `int` und `long` gilt:

- Wenn nicht explizit ein anderer Typ angegeben wurde, wird für alle Berechnungen intern `int` verwendet.
- Negative Zahlen werden (intern) als 2-Komplement gespeichert.
- Es findet keine Überlaufprüfung statt! Als Seiteneffekt gilt daher:

```
Integer.MAX_VALUE + 1 = Integer.MIN_VALUE
Integer.MIN_VALUE - 1 = Integer.MAX_VALUE
```

Für `byte` und `short` müssen Ausdrücke in vielen Rechenfunktionen explizit gecastet werden, um eine Compiler-Warnung zu vermeiden.

```
byte b = (byte)( 3 + 5 )
```

4.2 Fließkommatypen

Für die Typen `float` und `double` findet eine Überlaufprüfung statt. In diesem Fall bekommt die Variable einen speziellen, reservierten Wert ‚positiv unendlich' („`POSITIVE_INFINITY`**"),** bzw. ‚negativ unendlich' („`NEGATIVE_INFINITY`") zugewiesen. So z.B. als Ergebnis einer Division durch Null.

Zusätzlich kennen diese Typen noch die speziellen Werte „`NaN`" (‚not a number' = ‚keine [gültige] Zahl') und sowohl eine *positive* als auch eine *negative* Null!

```
float a = 0.0F;
float b = -0.0F;
```

Wird kein Typensuffix angegeben, wird normalerweise `double` as Typ angenommen. Entsprechend wird normalerweise für alle Fließkommaberechnungen intern `double` benutzt.

Achtung: Bei der Rechnung mit Fließkommavariablen kann es zu Rundungsfehlern kommen, die im Extremfall zu signifikanten Abweichungen vom korrekten Ergebnis führen können!

Zum Beispiel:

```
1.0 / 3 * 100000 * 3 / 100000 = 0.9999999999999999
```

* Laut Spezifikation sind `\uFFFF` und `\uFFFE` keine gültigen Unicode-Symbole. Das letzte nutzbare Zeichen ist folglich `\uFFFD`, was für das Ersatzzeichen für nicht darstellbare Symbole steht, und damit mitunter optisch nicht von den beiden zuvor erwähnten zu unterscheiden ist.

4.3 Strings

Java benutzt durchweg die *utf16*-Kodierung für Strings, d.h. es werden jeweils mindestens 2 Byte pro Zeichen verwendet. Für Symbole mit einem Unicode-Wert ab `\u10000` werden entsprechend 4 Bytes verwendet.

Um Nicht-ASCII-Zeichen (z.B. Umlaute) auf der Konsole darstellen zu können, muss diese auch auf einen dazu kompatiblen Zeichensatz eingestellt sein (z.B. *utf8).*

Der Datentyp „`char`“ kann ein einzelnes dieser Zeichen aufnehmen (jedoch keine Zeichen aus dem Bereich über `\uFFFF`!). Um Literale vom Typ „`char`“ von Strings zu unterscheiden wird das Zeichen im Sourcecode in einfache Anführungszeichen eingeschlossen:

```
char c = 'a'
String s = "a"
```

Nicht eingebbare Zeichen, oder solche, die eine besondere Bedeutung im Quellcode haben, können über die folgenden Escape-Sequenzen maskiert werden:

`\t`	Tabulator (ASCII-Code: 9)
`\n`	Zeilenvorschub (eng: *newline;* ASCII-Code `10` bzw. `0x0a`)
`\r`	Zeilenrücklauf (eng: carriage return; ASCII-Code `13` bzw. `0x0d`)
`\\`	Backslash (‚\‘)
`\"`	Anführungszeichen
`\'`	Apostroph

Darüber hinaus sind auch „**`\b`**“ (Rückschritt) und „**`\f`**“ (Seitenvorschub) vordefiniert. Außerdem kann jedes Unicode-Zeichen auch numerisch spezifiziert werden:

`\u`*`hhhh`*	mit *`hhhh`* = 4-stellige Hexadezimalzahl	*`\ooo`*	mit *`ooo`*= 3-stellige Oktalzahl

Eine vollständige Liste aller Unicode-Symbole findet man auf http://www.unicode.org/charts/

Hinweis: im Gegensatz zu den Namen der Basistypen (s.o.) wird der Typenname „`String`“ immer mit einem Großbuchstaben geschrieben.

Strings (und Chars) können einfach mit dem Plus-Operator („+“) konkateniert werden. Weitere Funktionen sind in der Klasse „`String`“ zu finden. Einige Beispiele dazu im nächsten Abschnitt.

4.4 Arrays

Arrays eines bestimmten Types werden durch den Namen des Types gefolgt von (leeren) eckigen Klammern deklariert. Alternativ können die Klammern auch dem Namen der Variable folgen:

```
int[] intArray;   // deklariert ein Array vom Typ int
int intArray[];   // identisch mit der vorherigen Zeile.
```

Dabei ist die erste Variante ist jedoch die bevorzugte.

Das eigentliche Array wird durch ein ‚new‘ gefolgt vom Typenbezeichner und der Zahl der Elemente in eckigen Klammern erzeugt:

```
int[] intArray = new int[12];
```

Der Inhalt eines Arrays kann auch „inline“ in geschweiften Klammern vordefiniert werden:

```
String[] season = {"Winter", "Spring", "Summer", "Fall"};
```

Auf die Elemente eines Arrays kann sowohl lesend wie auch schreibend über die Klammer-Schreibweise zugegriffen werden. Dabei wird in eckigen Klammern der Index des Elementes angegeben, auf das zugegriffen werden soll:

```
String ses = season[0]; // der Variable ses wird „Winter“ zugewiesen
season[3] = "Autumn";   // das vierte Feld in season wird ersetzt.
```

Java kennt keine eigentlichen mehrdimensionalen Arrays; Es ist jedoch möglich, Arrays von Arrays, u.s.w., anzulegen:

```
int[][] intArray2D;     // Array von int-Arrays
int[][][] intArray3D    // Array eines Arrays von int-Arrays
```

Als Nebeneffekt können Arrays daher auch „fringed“ sein, d.h. sie müssen nicht unbedingt rechteckig bzw. quaderförmig angelegt werden.

Das erste Element eines Arrays hat immer die Indexnummer `0` (Null).

Jedes Array hat eine implizite Eigenschaft „*length*“, welches die Anzahl der Elemente in diesem Array enthält. Da die Zählung immer mit Null beginnt, ist die Indexnummer des letzten Elementes immer *length-1*. z.B. für eine Schleife über ein Array vom Typ ‚`char[]`‘ schreibt man:

```
for (int i=0; i<myArray.length; i++) {
      char it = myArray[i];
      ...
}
```

Da *length* keine Methode, sondern eine Eigenschaft ist, hat die Verwendung in der Schleifenbedingung, keine (oder kaum) negativen Auswirkungen auf die Laufzeit der Schleife.

Arrays sind grundsätzlich iterierbar; daher kann man auf die Elemente auch mittels einer Iterationsschleife (s.u.) zugreifen. Das obige Beispiel kann man daher auch wie folgt schreiben:

```
for (char it : myArray) { ... }
```

4.5 Enum-Variablen

Ein besonderer Variablentyp ist „enum“. Dieser besteht aus einer frei definierbaren (endlichen) Werteliste, und garantiert, dass die Variable stets einen der so definierten Werte annimmt.

Um einen eigenen `enum`-Typ zu definieren:

```
enum Jahreszeit {fruehling, sommer, herbst, winter};
```

Eine Variable kann dann wie folgt deklariert werden:

```
Jahreszeit season = winter;
```

Auch Enum-Variablen können als Iterator angesprochen werden (mittels „`.values()`“) und können daher in Iterationsschleifen verwendet werden.

4.6 Deklaration von Variablen und Konstanten

Variablen werden im Quelltext wie folgt deklariert:

```
Typenname Variablenname = Initialisierung;
```

dabei ist die *Initialisierung* optional aber (meist) empfehlenswert. Zur Initialisierung kann jeder Ausdruck verwendet werden, der einen Wert vom passenden Typ zurückgibt.

Das Schlüsselwort „`const`“ ist in Java zwar reserviert, aber es wird (derzeit) nicht benutzt.
Eine Art Pseudo-Konstante kann aber wie folgt deklariert werden:

```
final Typenname Variablenname = Initialisierung;
```

der einzige Unterschied zur normalen Variable ist hier das Schlüsselwort „`final`“. Das Weglassen der Initialisierung ist so zwar auch möglich, aber offenkundig nicht sehr sinnvoll.

Es ist üblich (wenn auch nicht wirklich notwendig), Konstanten einen Namen zu geben, der vollständig aus Großbuchstaben besteht.

Deklarationen können prinzipiell an jeder Stelle innerhalb einer Klasse oder Methode erscheinen. Der Gültigkeitsbereich („scope“) erstreckt sich aber prinzipiell nur über den Block, in dem die Deklaration steht (einschließlich untergeordneter Blöcke).

5. Verzweigungen

5.1 Die ‚`if`‘-Anweisung

Eine einfache if-Struktur hat die folgende Form:

```
if (boolscher-Ausdruck) {
     Anweisungen 1;
}
 else {
     Anweisungen 2;
}
```

Dabei ist die `else`-Anweisung (samt zweitem Anweisungsblock) optional, d.h. sie kann komplett entfallen, falls sie nicht benötigt wird.

Im Prinzip sind die geschweiften Klammern auch optional, und könnten entfallen, falls nur eine einzige Anweisung folgt, aber dies gilt als schlechter Programmierstil, und führt leicht zu (schwer auffindbaren) Fehlern.

Eine Ausnahme hiervon stellt die Konstruktion eines „`else if`“-Zweiges dar, bei dem es sich letztlich um eine neue „`if`“-Anweisung als einzigen Inhalt eines Umschließenden „`else`“- Zweiges handelt.

Die Bedinung „*boolscher-Ausdruck*“ muss unbedingt in Klammern stehen, und kann jeder beliebige Ausdruck sein, der einen boolschen Wert zurückgibt. Dazu gehören Vergleichsoperationen (s.u.), Methodenaufrufe oder auch Variable vom Typ „`bool`“.

Die wichtigsten Vergleichsoperationen:

==	Gleich*	**!=**	Ungleich
>	Größer	**>=**	Größer oder gleich
<	Kleiner	**<=**	Kleiner oder gleich

Mehrere Vergleichsbedinungen können über die folgenden Operatoren verknüpft werden:

&&	Logisches UND	**||**	Logisches (inklusives) ODER

Außerdem kann prinzipiell jede Bedinung negiert werden, indem ihr ein Ausrufezeichen vorangestellt wird.

Obwohl die logischen Operatoren „**&&**“ und „**||**“ sehr niedrige Priorität haben (also immer erst nach den Vergleichen betrachtet werden), empfiehlt es sich, großzügig Klammern (und Leerräume) einzusetzen, damit auch komplexe Ausdrücke lesbar bleiben.

* Achtung: das einfache Gleichheitszeichen („=“) ist kein Vergleichs-, sondern ein Zuweisungsoperator! Da aber auch eine Zuweisung einen Rückgabewert hat, und dieser u.U. auch vom Typ „`boolean`“ sein kann, wird eine Verwechslung nicht unbedingt vom Compiler bemerkt, und kann zu schwer nachvollziehbarem Fehlverhalten führen!

```
if ((variable_1 >= variable_2) && (variable_1 <= max)) ...
```

Achtung: Wenn der rechte Ausdruck einer solchen Verknüpfung das Ergebnis nicht mehr ändern kann (z.B. wenn bei „`||`“ der linke bereits *true* ergibt), wird er nicht mehr betrachtet. Dies kann unerwünschte Nebeneffekte haben, z.B. wenn der zweite Ausdruck einen Funktionsaufruf enthält, der unbedingt ausgeführt werden soll!

Vorsicht übrigens beim Vergleich von Fließkommavariablen, da es hier zu Ungenauigkeiten kommen kann, die zu unerwarteten Ergebnissen führen.

5.2 Die ‚`switch`‘-Anweisung

Mehrfachauswahlen lassen sich zwar auch mit verschachtelten `if`-Anweisungen realisieren, mit „`switch`“ geht das aber meist etwas eleganter:

```
switch (Ausdruck) {
case Konstante 1:
      Anweisungen 1;
      break;
...
case Konstante n:
      Anweisungen n;
      break;
default:
      Ersatzanweisungen;
}
```

Der *Ausdruck* wird nur *einmal* ausgewertet, und muss einen `byte`, `char`, `short`, `int` oder `long`-Wert zurückgeben.

Achtung: Die Ausführung der Anweisungen endet *nicht* automatisch mit der nächsten „`case`“-Anweisung. Deswegen wird normalerweise „`break`“ benutzt, um aus dem „`switch`“-Block herauszuspringen; In bestimmten Fällen kann es jedoch auch sinnvoll sein, „`break`“ wegzulassen, um einen anderen Programmablauf zu erzielen.

Die optionale „`default`“-Anweisung dient als Sprungmarke für den Fall, dass es für den Wert von *Ausdruck* keine passende „`case`“-Anweisung gibt.

5.3 Der Konditionaloperator ‚`?:`‘

Eine alternative Form des Konditional kann wie folgt gebildet werden:

```
( boolscher-Ausdruck ? Wert_1 : Wert_2 )
```

Im Unterschied zu ‚`if`‘ dient diese Konstruktion dazu, einen Wert zurückzugeben, und kann daher in Zuweisungen, Konkatenierungen o.ä. verwendet werden.

Falls der boolsche Ausdruck ‚`true`‘ zurückgibt, ergibt der gesamte Ausdruck den ersten Wert, andernfalls den zweiten.

Da Formeln mit diesem Operator jedoch schnell komplizitert und unübersichtlich werden, sollte man diesen Operator nur für relativ einfache Konstruktionen verwenden.

Die Klammern um den gesamten Ausdruck sind zwar nicht notwendig, helfen aber sehr dabei, solche Ausdrücke lesbar zu halten.

6. Schleifen

6.1 Abweisende Schleifen mit ‚`while`‘

Diese Schleifen werden so lange ausgeführt, bis eine Bedingung den Wert ‚`false`‘ zurückgibt. Da die Überprüfung vor dem Schleifenkörper steht, wird dieser nie ausgeführt, falls die Bedingung von Anfang an ‚`false`‘ ergibt.

```
while (Bedingung) {
      Schleifenkörper
}
```

6.2 Nicht-abweisende Schleifen mit ‚`do…while`‘

Dieser Schleifentyp funktioniert im Prinzip genau wie die einfache ‚`while`‘-Schleife, außer dass hier die Bedingung *nach* dem Schleifenkörper steht. Daher wird dieser auf jeden Fall mindestens einmal durchlaufen.

```
do {
      Schleifenkörper
} while (Bedingung)
```

6.3 Zählschleifen mit ‚`for`‘

Die Zählschleife dient dazu eine zuvor festgelegte Anzahl von Schleifendurchläufen abzuarbeiten. Allgemein ausgedrückt sieht sie wie folgt aus:

```
for (Initialisierung; Schleifenbedingung; Inkrementierung) {
      Schleifenkörper
}
```

Die Anweisung(en) im Feld *Initialisierung* werden *einmalig* zu Begin der Bearbeitung dieser Schleifenkonstruktion abgearbeitet. Es wird typischerweise dazu benutzt, eine oder mehrere Variablen auf einen Anfangszustand zu initialisieren. Mehrere Anweisungen werden durch Kommata getrennt. Man kann an dieser Stelle auch Variablen deklarieren; Diese sind dann nur innerhalb der ‚`for`‘-Schleife (einschließlich Schleifenkopf!) gültig. Beipiel: „`int i=0`“

Die *Schleifenbedingung* ist ein Ausdruck, der einen ‚`boolean`‘-Wert zurückgibt. Dieser Ausdruck wird *bei jedem Schleifendurchlauf* erneut ausgewertet; Ist der Rückgabewert ‚`false`‘, wird die Schleife abgebrochen. Zum Beispiel: „`i<=10`“

Das *Inkrementierung*s-Feld schließlich wird *am Ende jedes Schleifendurchlaufes* aufgerufen. Typischerweise wird man hier die benutzten Schleifenvariablen zu in- oder dekrementieren. Auch hier können mehrere Anweisungen durch Kommata getrennt angegeben werden. Beispiel: „`i++`“.

Der Vorteil in der Verwendung des Inkrementierungs-Feldes im Gegensatz zu einer Anweisung am Ende des Schleifenkörpers ist, dass erstere auch nach einer ‚`continue`‘-Anweisung auf jeden Fall ausgeführt werden.

6.4 Iteratorschleifen mit ‚`for`‘

Durch eine alternative Syntax der ‚`for`‘-Schleife kann man eine Iteratorschleife implementieren:

```
for (Variable : Iteratorobjekt) {
      Schleifenkörper
}
```

Hierbei wird der Schleifenkörper einmal für jedes Element des Iteratorobjektes aufgerufen; *Variable* enthält dabei jeweils ein anderes dieser Elemente. Dies funktioniert u.a. mit Arrays oder Listen.

6.5 Allgemeines zu Schleifen

Bei allen Schleifentypen können die geschweiften Klammern wegfallen, wenn der Schleifenkörper nur aus einer einzigen Anweisung besteht; und wie schon bei der ‚`if`'-Anweisung erwähnt, halte ich das für schlechten Programmierstil!

Wie schon bei ‚`switch`' gesehen, ist es mit der Anweisung ‚`break`' möglich, zu jedem Zeitpunk den gerade aktuelle Block (z.B. in einer Schleife) zu verlassen, ausgenommen nur ein Anweisungsblock in einem ‚`if`'-Konstrukt. Dadurch ist es auch möglich, einen *bedingten Abbruch* durchzuführen.

Etwas ähnliches macht die Anweisung ‚`continue`'; Allerdings springt sie nicht aus dem Block *heraus,* sondern nur an dessen Ende, sodass erneut die Überprüfung der Abbruchbedinung (und evtl. eine Inkrementierung, etc.) stattfinden kann.

Beiden Anweisungen kann als optionaler Parameter eine Sprungmarke übergeben werden, die angibt, für welche Schleife der Befehl gilt (ohne Parameter ist es die jeweils innerste).

7. Ausnahmebehandlung („exceptions")

Im Falle einer Programmausnahme (z.B. bei einem Laufzeitfehler) wird von der Java-Laufzeitumgebung ein „`exception`"-Objekt erzeugt und dem Programm zur Verfügung gestellt („geworfen").

7.1 Auffangen von Exceptions

Um Exceptions „aufzufangen" kann ein Programmblock von „`try`"- und „`catch`"- Anweisungen umschlossen werden. Letztere erhält als Parameter den Exception-Typen, der aufgefangen werden soll, sowie einen Variablennamen, unter dem das gefangene Objekt referenziert werden kann.

```
try {
      int i = 1/0;
      System.out.println("Dies wird nie ausgeführt!");
}
catch (ArithmeticException e) {
      System.out.println("Exception: " + e.getMessage());
}
```

Obwohl es möglich ist, einfach die relativ unspezifische Klasse „Exception" als Typ anzugeben, sollte man meist einen möglichst spezifischen Typen angeben, damit nicht die falsche Exception eingefangen wird.

Um auf unterschiedliche Exceptions passend reagieren zu können, ist es möglich, mehrere „`catch`"-Blöcke für unterschiedliche Exception-Typen anzugeben, die dann je nach Typ der geworfenen Exception benutzt werden.

Außerdem kann optional ein weiterer Block („`finally`") spezifiziert werden, der auf jeden Fall ausgeführt wird, ganz gleich, wie der „`try`"-Block verlassen wird, also auch im Fall, dass es keinen passenden „`catch`"-Block gibt. Tatsächlich ist es noch nicht einmal notwendig, überhaupt einen „`catch`"-Block zu deklarieren, wenn es stattdessen einen „`finally`"-Block gibt.

Wird in der aktuellen Methode keine passende ‚`catch`'-Anweisung gefunden, springt der Programmablauf eine Ebene ‚höher', zur aufrufenden Methode, u.s.w. bis zur ‚`main`'-Methode, mit der das Programm gestartet wurde.

Falls die Exception bis (spätestens) dorthin nicht aufgefangen wurde, bricht das Programm mit einer Fehlermeldung ab.

Um selbst eine Ausnahmesituation auszulösen, „wirft“ man eine neue Instanz der gewünschten Exception mittels des Befehls „`throw`“:

```
throw new MyException("Fehler: Benutzer inkompatibel.");
```

Eine Methode, die auf diese Weise eine Exception werfen kann, muss in der Methodendeklaration (s.u.) angeben, welche Exceptions generiert werden können. Mehr dazu im nächsten Kapitel.

8. Methoden

Unter Java können Methoden nur auf Klassenebene definiert werden. Prinzipiell haben alle Methoden einen Rückgabetyp (sind also mithin eigentlich Funktionen); Falls kein Rückgabewert benötigt wird, kann hier einfach „`void`“ angegeben werden.

```
<Zugriffsrechte> <Rückgabetyp> <Methodenname> (<Parameterliste>) {
        Methodenkörper;
        return <Rückgabewert>;
}
```

Die *Zugriffsrechte* können entweder als ‚öffentlich‘ („`public`“), ‚geschützt‘ („`protected`“) oder ‚privat‘ („`private`“) deklariert werden (mehr dazu später im Kapitel über Klassen). Zusätzlich kann man die Funktion noch als ‚statisch‘ („`static`“) deklarieren. Dazu mehr im Kapitel über Klassen.

Als Rückgabewert kann jeder Java-Datentyp (elementar oder Klasse) dienen, oder eben „`void`“, falls kein Rückgabewert benötig wird.

Der Rückgabewert, der „`return`“ übergeben wird, muss zu dem im Methodenkopf deklarierten Typen passen. Falls „`void`“ deklariert wurde, kann der Rückgabewert auch weggelassen werden.

Die Anweisung „`return`“ kann an jeder Stelle in der Methode aufgerufen werden (nicht nur an deren Ende). Tatsächlich kann diese Anweisung sogar ganz weggelassen werden, wenn der Rückgabetyp „`void`“ ist und der Rücksprung ohnehin am Ende der Methode erfolgen soll.

Als *Parameterliste* können beliebig viele Variablendeklarationen durch Kommata getrennt aufgelistet werden. Selbstverständlich ist auch eine leere Parameterliste möglich. Wichtig ist, dass beim Aufruf der Methode die Zahl und Reihenfolge der Parameter genau eingehalten weden müssen.

Als *Methodenname* kann jeder gültige Bezeichner (der kein reserviertes Schlüsselwort ist) verwendet werden. Methoden können auch *überladen* werden, d.h. der gleiche Methodenname kann mehrfach vorkommen, solange die gesamte *Signatur* (Methodenname plus Parametertypen) eindeutig ist.

Wie im Kapitel über „Exceptions“ gezeigt muss eine Methode im Methodenkopf anzeigen, ob und welche Exception sie möglicherweise ‚werfen‘ kann. Dies geschieht mittels des Schlüsselwortes „`throws`“:

```
... <Methodenname> (...) throws <Exception-Liste> { ...
```

Es können wie hier mehrere Exceptions durch Kommata getrennt aufgelistet werden.

9. Klassen

Klassen sind Beschreibungen von Objekten, welche wiederum als Bausteine für Programme dienen. In Java, als einer objektorientierten Programmiersprache, gibt es prinzipiell keine klassenfreien Methoden; Alles muss stets innerhalb von Klassen definiert werden.

9.1 Definition

Die üblichste Syntax zur Definition einer Klasse ist recht simpel:

```
class <Klassenname> {
      ...
}
```

Für die Namensgebung einer Klasse gelten die gleichen Vorgaben wie für die Namen von Methoden, außer dass der Klassenname prinzipiell einmalig sein muss. Es ist auch sinnvoll, der Quellcodedatei, welche die Klasse enthält, den gleichen Namen zu geben (plus Dateisuffix „`java`“).

Anders als bei Funktionen, wo es keine inneren Funktionen geben kann, können Klassen auch *innerhalb* von anderen Klassen definiert werden – für diese gelten dann beim Zugriff auch die gleichen Regeln wie für Funktionen oder Eigenschaften, d.h. sie können als „`private`“, „`protected`“ oder „`public`“ definiert werden.

Um eine Instanz einer Klasse zu erstellen, benutzt man den Befehl „new“, ähnlich wie bereits bei den Arrays eingeführt – allerdings werden evtl. Parameter in *runden* Klammern mitgegeben:

```
MyClass my = new MyClass(param1, param2);
```

9.2 Konstruktoren

Bei der Erstellung einer Instanz eines Objektes (durch den Befehl „new“) wird zunächst die Konstruktormethode aufgerufen, diese trägt prinzipiell den gleichen Namen wie die Klasse.

Konstruktoren haben prinzipbedingt keinen Rückgabetyp (sie „geben“ ja die gerade erstellte Klasseninstanz zurück), und sie sollten in den meisten Fällen „`public`“, also ‚öffentlich‘ deklariert sein, damit sie auch von außerhalb der Klasse aufgerufen werden können (es gibt jedoch Ausnahmen!). Parameter, die für diese Methoden definiert sind, können (und müssen) bei der Erstellung der Instanz mit „new“ auch übergeben werden.

```
class myClass {
      public myClass(String name) {
            ... // Konstruktormethode
      }
}
```

9.3 Statische Methoden und Eigenschaften

Nicht jede mit einer Klasse assoziierte Methode oder Eigenschaft macht Sinn als Teil der Objektinstanz. Manchmal sollten sie auch unabhängig von Instanzen bestehen, z.B. die Anfangs erwähnte „`main`“-Methode.

Man nennt solche Elemente, die der Klasse, und nicht dem Objekt zugeordnet sind „statische“ Elemente. Wie bereits Eingangs gezeigt, benutzt man das Schlüsselwort „`static`“ bei der Deklaration. Da diese dann nicht der Instanz sondern der Klasse zugeordnet sind, benutzt man den Klassennamen zur Referenzierung, und nicht einen Variablennamen.

Statische Methoden können nur auf andere statische Methoden oder statische Eigenschaften zugreifen. Umgekehrt können nicht-statische Methoden auch auf statische Elemente zugreifen.

Zugriffsrechte

Wie schon im Kapitel über Methoden erwähnt gibt es drei Schlüsselwörter, welche die Zugriffsberechtigungen für Methoden und Eigenschaften modifizieren. Diese sind „`public`“, „`protected`“ und „`private`“. Wird keine Zugriffsberechtigung angegeben, wird immer „`private`“ angenommen.

Ein Zugriffsrecht von „`private`“ bedeutet, dass auf die fragliche Methode oder Eigenschaft nur von der Klasse selbst zugegriffen werden kann. Bei „`public`“ dagegen kann man von außerhalb darauf zugreifen. Die dritte Option, „`protected`“ (‚geschützt‘) verhält sich gegenüber fremden Klassen wie „`private`“, und gegenüber abgeleiteten Klassen (siehe auch „Vererbung“) wie „`public`“.

Vererbung

Eine neue Klasse kann von *einer* anderen, bereits existierenden Klasse abgeleitet werden, solange diese nicht explizit als „`final`“ deklariert wurde. Man spricht dann von letzterer als der „Superklasse“ der neuen. Zu diesem Zweck dient das Schlüsselwort „extends“ in der Klassendefinition:

```
class <Klassenname> extends <Name der Superklasse> { ...
```

Bei dieser Ableitung ‚erbt‘ die neue Klasse alle Methoden und Eigenschaften der Ausgangsklasse, kann jedoch direkt nur auf solche zugreifen, die entweder „`public`“ oder „`protected`“ deklariert wurden.

Prinzipiell kann die neue Klasse Methoden (oder auch Eigenschaften) der Superklasse überschreiben (engl. „overriding“) um damit das Verhalten der abgeleiteten Klasse an deren Erfordernisse anzupassen.

Dies ist zwar prinzipiell auch mit Klasseneigenschaften möglich, aber dies ist natürlich nur sinnvoll, wenn auf diese Weise der Typ einer Eigenschaft verändert werden soll. Da dies dann aber häufig zu schwer zu durchschauenden Nebeneffekten führen kann, sollte man dies nur in besonderen Ausnahmefällen machen.

Zugriff auf die übergeordnete Klasse

Eine abgeleitete Klasse kann mittels des Schlüsselwortes „`super`“ explizit auf die Superklasse zugreifen. Dies ist v.a. hilfreich, um auf überschriebene Methoden zugreifen zu können, oder um, wie im folgenden Beispiel, den Konstruktor der Superklasse aufrufen zu können:

```
super();          // Aufruf des Konstruktors der Superklasse
super.test();     // Aufruf der Methode „Test“ der Superklasse
```

Die „main“-Methode

Eine besondere Bedeutung hat auch die „`main`“-Methode, wie sie auch in dem Beispiel auf der ersten Seite benutzt wird: Diese muss statisch und öffentlich deklariert werden, und dient dann als Einstiegspunkt für den Programmstart.

Dies dient vor allem dazu, ein Java-Programm überhaupt starten zu können, kann aber auch zum Testen von Programmkomponenten hilfreich sein, indem man nämlich die „`main`“-Methode so implementiert, dass ihr direkter Aufruf die korrekte Funktionsweise der Klasse austestet.

Übersicht über die Klassenbibliothek

Neben einer Programmiersprache enthält Java auch ein komplexes Framework, das zahlreiche häufig benötigte Funktionen abdeckt. Hier wird nur eine kurze Auswahl der allerwichtigsten Klassen und nur die wichtigsten ihrer Methoden vorgestellt.

Merke: Konstruktoren werden bei der Erstellung eines neuen Objektes mit „new“ aufgerufen. Statische Methoden werden an den Klassennamen angehängt (z.B. „`Arrays.sort(arr)`“) während Instanzmethoden an den Variablennamen einer Instanz angehängt werden (z.B. „`s.charAt(12)`“).

java.lang.String

Konstruktor **`String`**`(char[] chars)`

erzeugt einen neuen String aus einem Array vom Typ „`char[]`“. Beispiel:

```
char[] chars = {'a', 'b', 'c'};
String txt = new String(chars);    // txt hat jetzt den Wert „abc“.
```

Es existieren zahlreiche ähnliche Konstruktoren, die Arrays vom Typ ‚`int`‘, ‚`byte`‘ oder auch StringBuffer- und StringBuilder-Objekte als Parameter annehmen.

`char` **`charAt`**`(int index)`

gibt das Zeichen an der Position `index` im String als `char` zurück.

`int` **`compareTo`**`(String str)` *oder*
`int` **`compareToIgnoreCase`**`(String str)`

Vergleichen den Inhalt des Strings mit dem eines anderen. Das Ergebnis ist `0` wenn die beiden Strings identisch sind, `1` wenn der Vergleichsstring lexigraphisch vor dem anderen einzuordnen ist, und `-1` wenn er danach einzuordnen ist.

`int` **`indexOf`**`(String substr)` *oder*
`int` **`indexOf`**`(String substr, int startPos)`

Gibt die Index-Position (`0`-basiert) des ersten Vorkommens des angegebenen Substrings in dem String an. Falls der Substring nicht darin vorkommt, wird `-1` zurückgegeben.

`int` **`lastIndexOf`**`(String substr)` *oder*
`int` **`lastIndexOf`**`(String substr, int startPos)`

Wie `indexOf`, es wird jedoch das *letzte* Vorkommen des Substrings gesucht.

`int` **`length`**`()`

Gibt die Länge des Strings in Unicode-Zeichen zurück. Da Index-Positionen `0`-basiert sind, entspricht `length()-1` immer der letzten Index-Position in einem String.

`boolean` **`matches`**`(String regex)`

Gibt `true` zurück, falls der String durch den übergebenen *regulären Ausdruck* beschrieben wird.

`String` **`replaceAll`**`(String regex, String ersatz)`

Ersetzt alle Treffer für den übergebenen *regulären Ausdruck* durch den Ersatzstring.

`String[] **split**(String regex)` *oder*
`String[] **split**(String regex, int limit)`

Erstellt ein Array aus Teilstücken des Strings, die jeweils an Treffern des regulären Ausdruckes aufgespalten werden. Optional kann ein Maximum an Teilstücken angegeben werden.

`char[] **toCharArray**()`

Konvertiert den String in ein Array vom Typ char, welches die einzelnen Buchstaben als Elemente enthält.

`String **toLowerCase()**` *oder*
`String **toUpperCase()**`

Erstellt einen neuen String der den gleichen Text wie der Ausgangsstring enthält, wandelt jedoch alle Großbuchstaben in Kleinbuchstaben um (`toLowerCase`) bzw. umgekehrt (`toUpperCase`).

`String **trim**()`

Erstellt einen neuen String, der dem Ausgangsstring entspricht, dem aber führende oder nachfolgende Whitespaces (Leerzeichen, Tabs, etc.) fehlen.

`boolean **startsWith**(String prefix)` *oder*
`boolean **endsWith**(String suffix)`

Ergibt `true` wenn der String mit dem angegebenen Teilstring beginnt bzw. endet.

java.lang.Math *und* java.lang.StrictMath

Diese Klassen enthalten eine Reihe von Methoden für elementare mathematische Berechnungen. Beide Klassen enthalten jeweils die gleichen Methoden, jedoch wurde „`Math`“ vor allem auf Geschwindigkeit optimiert, was bei Fließkommaberechnungen mituner zu Lasten der Genauigkeit gehen kann. Alle Methoden und Konstanten dieser Klassen sind statisch. Für viele dieser Funktionen gibt es Varianten für „float“ und andere nummerische Datentypen.

`static double **E**,`
`static double **PI**`

Konstanten, welche die mathematischen Werte *e* bzw. π jeweils bestmöglich abbilden.

`static double **ceil**(double a),`
`static double **floor**(double a)`

Gibt den nächsten ganzzahligen Wert größer („`ceil`“) bzw. kleiner („`floor`“) als der übergebene Wert zurück.

`static long **round**(double a),`
`static double **rint**(double a)`

Runden den übergebenen Parameter zur nächsten Ganzzahl auf- oder ab. Der Unterschied zwischen diesen beiden Funktionen ist das Verhalten wenn der Nachkommateil genau *0,5* beträgt: „`round`“ rundet in diesem Fall zur nächsthöheren Zahl auf, „`rint`“ nimmt immer die gerade Nachbarzahl.

`static double **signum**(double d)`

Ergibt `+1` falls der Parameter positiv ist, `-1` falls er negativ ist, und `0` falls `0`.

`static float` **`abs`**`(double a)`

Ergibt den Abstand der Zahl *a* von `0.0` – falls die Zahl positiv ist, entspricht dies der übergebenen Zahl, falls sie negativ ist, entspricht das Ergebnis *-a*.

`static double` **`max`**`(double a, double b)` *und*
`static double` **`min`**`(double a, double b)`

Ergibt die größere („`max`“) bzw. die kleinere („`min`“) der beiden übergebenen Zahlen.

`static double` **`pow`**`(double x, double y)` *(= x^y),*
`static double` **`sqrt`**`(double x)` *(= $\sqrt{x}$),*
`static double` **`exp`**`(double x)` *(= e^x),*
`static double` **`log`**`(double x)` *(= ln x),*
`static double` **`log10`**`(double x)` *(= log_{10} x),*
`static double` **`sin`**`(double x)` *(= sin x),*
`static double` **`cos`**`(double x)` *(= cos x),*
`static double` **`tan`**`(double x)` *(= tan x)*

Verschiedene Funktionen, entsprechend ihrer mathematischen Equivalente.

`static double` **`random`**`()`

Erzeugt eine (pseudo-) zufällige Zahl im Bereich zwischen (einschließlich) `0.0` und (ausschließlich) `1.0`.

Anwendungsbeispiel:

Um eine mathematische Formel wie die folgende umzusetzen

$$x = \sqrt{\frac{\ln(y+1)}{e^y}}$$

schreibt man in Java:

```
x = Math.sqrt(Math.log(y+1)/Math.exp(y))
```

java.util.Arrays

`static void` **`sort`**`(int[] arr)` *oder*
`static void` **`sort`**`(int[] arr, int anfang, int ende)`

Sortiert das angegebene Array aufsteigend nach dem Wert der Arrayelemente. Optional kann ein Bereich angegeben werden, der sortiert werden soll; Dieser wird einschließlich der Position ‚`anfang`‘ und ausschließlich ‚`ende`‘ definiert.

Anstelle von ‚`int`‘-Arrays können auch solche aller Elementartypen sortiert werden; außerdem von allen Objekten, welche das ‚`Comparable<T>`‘-Interface implementieren (z.B. „`String`“, „`Timestamp`“, „`URI`“, etc.).

Zur Sortierung wird für die Elementartypen eine modifizierte QuickSort-Variante, für Objekte ein modifiziertes MergeSort verwendet.

`static int` **`binarySearch`**`(int[] arr, int key)` *oder*
`static int` **`binarySearch`**`(int[] arr, int anfang, int ende, int key)`

Führt eine binäre Suche nach dem Wert ‚`key`' im Array durch. Hierfür muss das Array aufsteigend sortiert vorliegen (z.B. durch die Methode ‚`sort`"), ansonsten ist das Ergebnis nicht vorhersehbar.

Falls der Wert im Array gefunden wird, wird der Index des Elementes zurückgegeben. Falls mehrere Elemente mit dem gleichen Wert vorhanden sind, ist nicht vorhersagbar, welches von diesen gefunden wird.

Falls der Wert nicht gefunden wird, wird die Position zurückgegeben, an der ein neues Element mit diesem Wert eingefügt werden müsste, um die Sortierung beizubehalten, und zwar in der Form „`(-(`*`Einfügeposition`*`) – 1)`".

Neben ‚`int`'-Arrays können dieser Funktion auch Arrays aller Elementartypen übergeben werden; außerdem auch Arrays von Objekten, welche das ‚`Comparable<T>`'-Interface implementieren (z.B. „`String`", „`Timestamp`", „`URI`", etc.).

`static boolean` **`equals`**`(int[] arr1, int[] arr2)`

Ergibt `true` falls die beiden übergebenen Arrays gleich sind. Sie gelten als gleich, wenn sie über die gleiche Anzahl von Elementen verfügen, und die gleichen Werte an den gleichen Positionen stehen.

Wie üblich können nicht nur ‚`int`'-Arrays verglichen werden, sondern solche aller Elementartypen und auch beliebiger Objekte.

`static boolean` **`deepEquals`**`(Object[] arr1, Object[] arr2)`

Ähnlich wie ‚`equals`' für Objekte, aber es wird ein „tiefer" Vergleich (d.h. einschließlich beliebig tief verschachtelter Objekte) durchgeführt.

`static void` **`fill`**`(int[] arr, int wert)` *oder*
`static void` **`fill`**`(int[] arr, int anfang, int ende, int wert)`

Weist jedem Element in einem Array den angegebenen Wert zu. Falls ein Anfangs- und Endwert angegenen wurde, wird nur den Elemente von (einschließlich) ‚`anfang`' bis (ausschließlich) ‚`ende`' der Wert zugewiesen.

Außer für‚`int`'-Arrays kann diese Funktion auch für Arrays aller Elementartypen sowie beliebiger Objekte verwendet werden