# Kapitel 6: Konzepte imperativer Sprachen

Imperative Sprachen waren historisch die ersten „höheren Programmiersprachen“ (3. Generation). Da Paradigma imperativer Sprachen (Programmieren als strukturierte Folge von Speicheränderungen) entspricht dem Konzept der von-Neumann-Architektur. Das Basiskonzept der imperativen Programmierung besteht darin, dass Anweisungen den Wert von Speicherzellen (Variablen) verändern. Strukturierungskonzepte sind die Gruppierung von Anweisungen in Anweisungsblöcken, Fallunterscheidungs- und Wiederholungsanweisungen, und Prozeduren oder Funktionen.

## 6.1 Variablen, Datentypen, Ausdrücke

Betrachten wir als Beispiel zwei Java-Implementierungen der Fakultätsfunktion:

```
static int fakRek(int x) {
      return (x<=0) ? 1 : x * fakRek(x-1);
}

static int fakIt(int x) {
      int result = 1;
      while (x>0) { result *= x--; };
      return result;
}
```

Typische Sprachelemente imperativer Sprachen, die hier auftreten, sind Methodendeklarationen und Methodenaufrufe, Parameter und lokale Variablen, Terme mit arithmetischen und logischen Operatoren, Zuweisungen und Dekremente, bedingte Terme bzw. Fallunterscheidungen, und Wiederholungsanweisungen. Die einzelnen Sprachelemente werden nachfolgend erläutert. Für weitere Informationen verweisen wir auf den „Mini-Javakurs“ unter http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html.
*Variablen* haben in typisierten Sprachen wie Java immer einen festgelegten Datentyp. Jede Variable entspricht einer Speicherzelle. Dies gilt natürlich nur, falls die Variable einen Typ hat, der durch ein Speicherwort repräsentiert werden kann; eine Variable eines komplexen Typs enthält einen Verweis auf ein Objekt (Gruppe von Speicherzellen). Es gibt mehrere Arten von Variablen: Methoden-Parameter, lokale Variablen, Instanzvariablen und Klassenvariablen.

- Methoden haben *Parameter* (Ein/Ausgabevariablen)
- Methoden können *lokale Variablen* haben
- Jedes Objekt hat *Instanzvariablen* (nicht statisch)
  (Das Wort „Instanz“ ist eine schlechte Übersetzung des englischen „instance“, die sich aber nichtsdestotrotz eingebürgert hat.)
- Klassen können *Klassenvariablen* besitzen (statisch)

Instanzvariablen und Klassenvariablen heißen auch die *Datenfelder* der Klasse. Betrachten wir ein (willkürliches) Beispiel:

```
public class Main {
  static double wechselkurs = 1.32;
  int betrag = 7;
  static int zehnCentStücke(double p) {
    int cent = (int)((p-(int)(p)) * 100);
    return((cent==20) ? 1 : 0) ;}
  public static void main(String[] args) {
    double p = 2.70;
    System.out.print("p= ");
    System.out.println(p);
  }
```

```
}
```

Hier ist `p` in `main` eine lokale Variable, `p` in `zehnCentStücke` ein Parameter, `betrag` eine Instanzvariable und `.wechselkurs` eine Klassenvariable.
Java kennt folgende primitive Datentypen:

- *Zahlen*: int, byte, short, long, float, double (43.8F, 2.4e5)
- *Zeichen*: char (z.B. `'x'`) Sonderzeichen `'\n'`, `'\"'`, `'\\'`, `'\uFFF0'`
- *Zeichenreihen* (Strings): "Infor" + "matik"
  (streng formal sind Zeichenreihen keine primitiven Objekte, obwohl man sie als Konstante aufschreiben kann)
- *Wahrheitswerte*: boolean (true, false)
- *„Leerer Typ"* void

Der leere Typ wird aus formalen Gründen gebraucht, wenn z.B. eine Methode kein Ergebnis liefert. Hier sind einige Beispiele von Datendeklarationen:

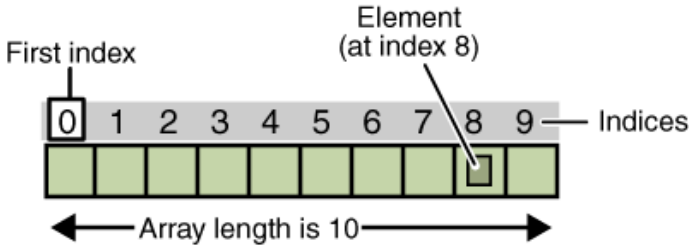
```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;

int decVal = 26;
int octVal = 032;
int hexVal = 0x1a;

double d1 = 123.4;
double d2 = 1.234e2;
float f1 = 123.4f;

String s = „ABCDE";
```

Benötigt man Variablen für mehrere gleichförmige Daten (d.h. viele Daten mit gleichem Typ), so kann man eine Reihung (Array) deklarieren (http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html).




Die Reihung könnte etwa wie im folgenden Beispielprogramm verwendet werden.

```
class ArrayDemo {
  public static void main(String[] args) {
    int[] anArray; // eine Reihung ganzer Zahlen
    anArray = new int[10]; // Speicherallokation
    anArray[8] = 500;
    System.out.println("Element at index 8: " + anArray[8]);
  }
}
```

Der Rumpf einer Java-Methode besteht aus einem *Block*, (Groovy: *closed block* oder *closure*) d.h. einer Folge von *Anweisungen* (statements). Bei der *Ausführung* werden diese der Reihe

nach abgearbeitet. Die wichtigste Art von Anweisungen sind *Ausdrücke* (expressions). Bei ihrer *Auswertung* liefern sie einen Wert ab und können per Nebeneffekt den Inhalt von Variablen verändern.
Hier ist ein Ausschnitt aus der Syntax von Java in BNF, in der Ausdrücke definiert werden (vgl. http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html, Kap. 18):

- *Expression::=Expr1 [AssignmentOperator Expr1]*
- *Expr1::=Expr2 [*? *Expression* : *Expr1]*
- *Expr2 ::=Expr3 {Infixop Expr3}*
- *Infixop::=* + | * | < | == | != | || | && | …
- *Expr3::= PrefixOp Expr3* |
  ( *Expr* | *Type* ) *Expr3* |
  *Primary {Selector} {PostfixOp}*
- *Prefixop::=* ++ | -- | ! | ~ |+ | -
- *Primary::=* ( *Expression* ) | *Literal* | new *Creator* |
  *Identifier* { . *Identifier* }[ *IdentifierSuffix]* | …

**ACHTUNG:** nach dieser Syntax ist eine *Zuweisung* also auch ein *Ausdruck*! Es stellt sich die Frage, was der Wert einer Zuweisung (etwa `x = 3`) ist: der Wert ist bei einer Zuweisung immer der zugewiesene Wert (d.h. das Ergebnis der rechten Seite, im Beispiel also `3`). Daher ist es möglich, eine Zuweisung auf der rechten Seite einer Zuweisung zu verwenden (also etwa `y = (x = 3)` oder `x = y = z`)
Beispielweise könnten Ausdrücke etwa wie folgt aussehen:

```
17 + 4
x = x + 1
b = b != !b
a += (a = 3)
v[i] != other.v[i]
cube[17][x+3][++y]
"Länge = " + ia.length + "," + ia[0].length + " Meter"
```

Java verlangt, dass jede lokale Variable vor ihrer Verwendung initialisiert wurde, und zwar so, dass es statisch erkennbar ist. Diese Bedingung wird vom Compiler mit einer so genannten *statischen Analyse* überprüft und in Eclipse ggf. bereits beim Tippen des Programmtextes als Fehler markiert. Allerdings kann die statische Analyse nur bedingt erfolgreich sein. Es gibt keinen Compiler, der in jedem Fall erkennt, ob alle Variablen korrekt initialisiert sind! Eine gute Faustregel ist es, alle Variablen gleich bei der Deklaration zu initialisieren.
In Java gibt es folgende Operatoren (vergleiche die oben angegebene Syntax):

- **Arithmetische Operatoren**: +, -, *, /, %, ++ und —
  / und % sind (ganzzahlige) Division und Rest
  ++ und — als Post- und Präfixoperatoren; a++ erhöht a um 1 und ergibt a, während ++a den Wert a+1 liefert
  + dient auch zur Konkatenation von Zeichenreihen!
- **Relationale Operatoren**: ==, !=, <, <=, > und >=
  Achtung! == und != vergleichen in Java bei Objekten nur Referenzen!
- **Logische Operatoren**: !, &&, ||, &, | und ^
  a || b wertet b nur aus falls a nicht gilt
- **Zuweisungsoperatoren** = , +=, -=, *=, /=, %, &=, |=, ^=, <<=, >>= und >>>=
- **Fallunterscheidung** (dreistellig) x? y: z

- **Bitoperatoren** ~, |, &, ^, >>, >>> und <<; Bitoperatoren können auch auf numerische Typen (int, long usw.) angewendet werden (siehe Bit-Repräsentation)
- Operatoren `new` und `instanceof`
  `a instanceof b` gibt an, ob Objekt `a` eine Instanz des Typs `b` oder einer ihrer Unterklassen ist ; in Groovy kann man das auch mit dem Attribut .class erreichen:
  ```
  int i=5; println i.class
  ```
- Operatoren für **Member**- und **Array-Zugriff**
  ```
  MeineKlasse.meinDatenfeld
  meinArray[7]
  ```

Hier sind noch drei Anmerkungen für Spezialisten:

```
public class Demo
{
  static public int sampleMethod() {
     int i = 0;
     boolean b = ++i==i++ | ++i==i++;
     return i;
  }
  static public int sampleMethod2() {
     int i = 0;
     boolean b = ++i==i++ || ++i==i++;
     return i;
  }
}
```

Das Ergebnis von `sampleMethod` und `sampleMethod2` unterscheidet sich, da im zweiten Fall der hintere Teil der Disjunktion nicht ausgewertet wird. *Merke:* Dies ist kein Beispiel für gute Programmierung! Man sagt, dass ein Ausdruck einen Nebeneffekt (oder auch Seiteneffekt) hat, wenn er neben dem eigentlichen Ergebnis weitere Variablen verändert. Ausdrücke mit Nebeneffekten sind schwer verständlich und sollten deshalb möglichst wenig verwendet werden!

```
public class Rechtsshift {
  static int i = -64;
  static int j = i>>2;
  static int k = i>>>2;
}
```

In diesem Beispiel hat i den Wert -64, j den Wert -16 und k den Wert 1073741808. Merke: Bit-Shifts sollten nur mit Vorsicht verwendet werden!

In Groovy gibt es die Möglichkeit simultaner Mehrfachzuweisungen:
```
def (a, b) = [1, 2]
```
ergibt a==1 und b==2.

Wie bereits oben erwähnt, sind Java und Groovy streng typisierte Sprachen. Das bedeutet, Alle Operatoren in Java sind strikt typisiert, d.h., es wird geprüft, ob der Typ dem verlangten entspricht. Z.B. kann << nicht auf `float` oder `String` angewendet werden. Zur Typumwandlung gibt es entsprechende Konversions- (Casting-)Operatoren; z.B.
```
(int)3.14 // ergibt 3
(double)3.14f // Ausweitung
```
Dabei ist zu beachten, dass bestimmte Typen inkompatibel sind
`(z.B. boolean und int)`

In Groovy wird der Typ einer Variablen, wenn er nicht vom Benutzer angegeben wurde, automatisch bestimmt. Dadurch können auch Konversionen automatisch berechnet werden. Es ist trotzdem ein guter Programmierstil, den Typ von Variablen statisch festzulegen und Konversionen explizit anzugeben.
Ausdrücke werden gemäß den „üblichen“ Operator-Präzedenz-Regeln ausgewertet.
Regeln für die Auswertungsreihenfolge sind:

- linker vor rechter Operator, dann Operation
- gemäß Klammerung
- gemäß Operator-Hierarchie

Achtung: Die Reihenfolge der Auswertung von Ausdrücken kann das Ergebnis beeinflussen!
Beispiel:

```
int i=2, j = (i=3)*i; // ergibt 9
int i=2, j = i*(i=3); // ergibt 6
```

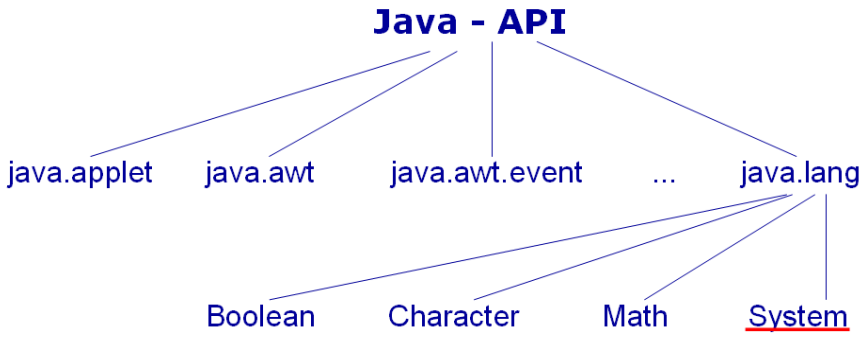
Überraschenderweise ist die Multiplikation also nicht kommutativ! Generell ist zu sagen, dass die Verwendung von Nebeneffekten schlechter Programmierstil ist!

## 6.2 Anweisungen und Kontrollstrukturen

In Java gibt es folgende Arten von Anweisungen:

- Leere Anweisung ;
- Block, d.h. eine mit {…} geklammerte Folge von Anweisungen; Blöcke dürfen auch geschachtelt werden
- Lokale Variablendeklaration
- Ausdrucksanweisung
  - Zuweisung
  - Inkrement und Dekrement
  - Methodenaufruf
  - Objekterzeugung

Ein wichtiges programmiersprachliches Mittel ist der Aufruf von Methoden der API (applications programming interface). Dazu muss die entsprechende Bibliothek importiert werden (falls sie es nicht schon standardmäßig ist): import java.awt.*.



(Grafik: © Bothe 2009)

Zugriff auf Bibliotheksfunktionen erfolgt mit Qualifizierern, z.B.

```
System.out. println(„Hallo“)
java.util.Date d = new java.util.Date ();
```

Zur Steuerung des Ablaufs werden Fallunterscheidungen verwendet.

- if-Anweisung
  - `if (ausdruck) anweisung; [else anweisung;]`
  - üblicherweise sind die Anweisungen Blöcke.
  - hängendes `else` gehört zum innersten `if`
    Beispiel: `if (b1) if (b2) a1 else a2`
    wird `a2` ausgeführt, wenn `b1` wahr und `b2` falsch ist?
- switch-Anweisung
  - ```
    switch (ausdruck) {
    {case constant : anweisung}
    [default: anweisung]}
    ```

D.h., in einer `switch`-Anweisung kommen beliebig viele `case`-Anweisungen, optional gefolgt von einer `default`-Anweisung. Bei der Auswertung wird zunächst der Ausdruck ausgewertet und dann in den entsprechenden Zweig verzweigt. Switch-Anweisungen sind tief geschachtelten if-Anweisungen vorzuziehen, da sie normalerweise übersichtlicher und effizienter sind.

**Kontrollflusselemente**

Folgende Wiederholungsanweisungen und Steuerungsbefehle werden häufig gebraucht:

- while-Schleife
  - allgemeine Form:
    ```
    while (ausdruck) anweisung;
    ```
    Die while-Schleife ist abweisend: Vor der Ausführung der Anweisung wird auf jeden Fall der Ausdruck geprüft. Beispiel:
    ```
    i=fak=1; while (i++<n) fak*=i;
    ```
  - In Java gibt es zusätzlich die nichtabweisende Form
    ```
    do anweisung; while (ausdruck);
    ```
    diese Schleife wird auf jeden Fall einmal durchlaufen, dann wird geprüft. Beispiel:
    ```
    do fak *= i++; while (i<=n)
    ```
    Jedes `do A while (B)` kann gleichwertig geschrieben werden als
    `A; while (B) do A;` (Achtung: Code-Replikation!)
- for-Schleife
  - allgemeine Form:
    ```
    for (init; test; update) anweisung;
    ```
    Beispiel:
    ```
    for (int i= fak = 1; i<=n; i++){ fak *= i;};
    ```
  - In Java darf die Initialisierung mehrere Anweisungen (mit Komma getrennt!) enthalten (auch gar keine); in Groovy wird an Hand der Zahl der Anweisungen entschieden, was Initialisierung, Test und Update ist (Benutzung nicht empfohlen!)
- continue, break und return
  - `continue` verlässt den Schleifendurchlauf
  - `return` verlässt die aufgerufene Methode
  - `break` verlässt die Schleife komplett
    Es kann auch zu einer markierten äußeren Schleife gesprungen werden:
    ```
    outer: while (A) {
          inner: while (B) {
                      dosomething;
                      if (C) break outer
                }
    }
    ```
    ```

In Groovy sind darüber hinaus folgende vereinfachte Schreibweisen erlaubt:

- `5.times{…}`
- `(1..5).each{fak *= it}` //Schleifenvariable 'it'
- `1.upto(5){fak *= it}`
- `for (i in [1,2,3,4,5]){fak *= i}`
- `for (i in 1..5){fak *= i}`
- `for( c in "abc" ){…}`

## 6.3 Sichtbarkeit und Lebensdauer von Variablen

Unter der *Sichtbarkeit* (Skopus) einer Variablen versteht man den Bereich im Quellcode, in dem dieser Variablenname verwendet werden darf. Der *Gültigkeitsbereich* ist derjenige Teil der Sichtbarkeit, in der man auf den Inhalt der Variablen zugreifen kann. Die *Lebensdauer* ist die Zeitspanne, während der für die Variable Speicherplatz reserviert werden muss
Für jede der verschiedenen Variablenarten gibt es detaillierte Festlegungen hierzu.
Beispiel:

```
public class xyz {
   int x;
   public xyz() {
        x = 1;
    }
    public int sampleMethod(int y) {
        int z = 2;
        return x + y + z;
    }
}
```

In diesem Beispiel bezieht sich `x` innerhalb von `xyz` und in `sampleMethod` auf die Instanzvariable, `y` in `sampleMethod` auf den Parameter, und `z` auf die lokale Variable.
Variablen können sich auch gegenseitig verschatten: Unter *Verschattung* versteht man die Einschränkung des Gültigkeitsbereichs einer Variablen durch eine andere Variable gleichen Namens. Es gelten folgende Regeln:

- Lokale Variablen werden in einem Block (einer Folge von Anweisungen) deklariert und sind nur für diesen verwendbar.
- Lokale Variable sind ab ihrer Deklaration bis zum Ende des Blocks oder der Methode sichtbar.
- Instanzvariable können von gleichnamigen lokalen Variablen verschattet werden, d.h., sie sind sichtbar aber nicht gültig.

Verschattung ist hilfreich, um das *Lokalitätsprinzip* zu unterstützen (z.B. in Laufanweisungen). Es ist nicht erlaubt, dass lokale Variablen sich gegenseitig oder Methodenparameter verschatten

Beispiel für Verschattung:

```
public class xyz {
 int x;

 public xyz() {
   x = 1;
 }

 public int sampleMethod(int y) {
   int x = 2;
   return x + y;
 }
}
```

Gültigkeitsbereich und Sichtbarkeit kann in Java/Groovy auch durch Modifikatoren eingeschränkt werden. Öffentliche (public) Klassenvariablen sind überall sichtbar; private Klassenvariablen (private) sind in der sie enthaltenden Klasse überall sichtbar, dienen also als „gemeinsamer Speicher“ der Objekte. Methodenvariablen sind nur innerhalb der Methode sichtbar, sind also „Hilfsvariablen“ zur Ausführung der Methode; der Speicherplatz wird nur während der Abarbeitung der Methode reserviert. Es gehört zum guten Programmierstil, alle Variablen so lokal wie möglich zu deklarieren, um die Gültigkeitsbereiche überschaubar zu halten.

Als Konvention hat sich eingebürgert, Variablennamen immer mit Kleinbuchstaben beginnen zu lassen und Teilworte mit Großbuchstaben direkt anzufügen. Beispiel:

```
int diesIstEineIntegerVariable.
```

Finale Variablen (Konstanten) werden in Großbuchstaben geschrieben, wobei Teilworte durch _ getrennt werden. Beispiel:

```
final float MAX_TEMPO = 5.0;
```

Methodennamen werden wie Variablennamen geschrieben und sollten Verben enthalten, Beispiel:

```
public void fahreRueckwaerts (float tempo);
```

Klassennamen beginnen mit Großbuchstaben und sollten substantivisch sein. Beispiel:

```
class FahrzeugReifen
```

Für weitere Informationen lesen Sie bitte die Java Coding Conventions von SUN unter http://java.sun.com/docs/codeconv/html/CodeConventionsTOC.doc.html.