Abschnitt 5.3

Objekte, Klassen, Kapselung

Objekte, Klassen, Kapselung

Die objektorientierte Modellierung bildet die Grundlage für die objektorientierte Implementierung eines Systems.

Dieser Abschnitt beschäftigt sich mit der Umsetzung der vorgestellten Modellierungskonzepte in objekt- orientierte Programmiersprachen, insbesondere Java

Überblick:

- Beschreibung von Objekten und Klassen
- Anwendung von Objekten
- Spracherweiterungen: Ausnahmebehandlung und Initialisierung
- Anwenden und Entwerfen von Klassen
- Spracherweiterungen: Überladen, Klassenattribute und Klassenmethoden
- Zusammenwirken der Spracherweiterungen
- Rekursive Klassen
- Typsystem von Java und parametrische Typen
- Kapselung und Strukturieren von Klassen

Unterabschnitt 5.3.1

Beschreibung von Objekten und Klassen

Beschreibung von Objekten und Klassen

In der objektorientierten Programmierung betrachtet man die Ausführung eines Programms als ein System kooperierender Objekte.

Grundsätzlich gibt es zwei Konzepte zur programmiersprachlichen Beschreibung von Objekten:

- ***Prototyp-Konzept:***
  Der Programmierer beschreibt direkt einzelne Objekte. Neue Objekte werden durch Klonen existierender Objekte und Verändern ihrer Eigenschaften zur Laufzeit erzeugt.
- ***Klassenkonzept:***
  Der Programmierer deklariert Klassen als Beschreibung der Eigenschaften, die Objekte dieser Klasse haben sollen. Die Programmiersprache ermöglicht es, zur Laufzeit Objekte der Klassen zu erzeugen, aber nicht, die Klassen zu verändern.

Wir betrachten hier nur das Klassenkonzept.

Beschreibung von Objekten und Klassen (2)

Die Eigenschaften und das Verhalten eines (programmiersprachlichen) Objekts ergeben sich aus seinen möglichen Zuständen und daraus, wie es auf Nachrichten reagiert.

Eine Objektbeschreibung - insbesondere eine Klassendeklaration - muss daher festlegen:

- welche Zustände ein Objekt annehmen kann,
- auf welche Nachrichten es reagieren kann und
- wie die Methoden aussehen, mit denen ein Objekt auf den Empfang von Nachrichten reagieren kann.

Die Menge der möglichen Zustände eines Objekts entspricht den Wertebereichen seiner Attribute.

Die Reaktionen eines Objekts auf eintreffende Nachrichten legen sein dynamisches Verhalten fest.

Beschreibung von Objekten und Klassen (3)

Eine einfache ***Klassendeklaration*** in Java hat folgende Bestandteile:

```
class Person {              Klassenname

  String name;              Attribut

  Person(String n) {        } Konstruktor
    this.name = n;
  }

  String getName() {        } Methode
    return this.name;
  }
}
```

Ein Java-Objekt kann genau auf die Nachrichten reagieren, für die Methoden in seiner Klasse deklariert sind oder für die es Methoden geerbt hat (vgl. Abschnitt 5.3).

Deklaration von Klassen

Klassenname

Direkt hinter dem Schlüsselwort `class` wird der Name der Klasse angegeben. Objekt einer Klasse K nennt man auch ***Instanzen*** oder *Ausprägungen* von K.

Der Klassenname wird gleichzeitig als Typname für die Objekte dieser Klasse verwendet (*Klassentyp*). Er kann im Programm dann wie elementare Typen (`int, float`, usw.) für die Deklaration von lokalen Variablen, Parametern und Rückgabewerten verwendet werden.

Deklaration von Klassen (2)

Beispiel:

```
Person eineMethode(Person p) {
  ...
  Person vater;
  ...
}
```

Außer den Objekten einer Klasse K gehören auch alle Objekte von Unterklassen von K zum Typ K (siehe Abschnitt 5.3).

Deklaration von Klassen (3)

Attribute

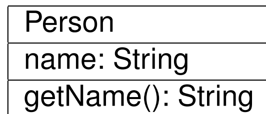
Innerhalb einer Klasse K können beliebig viele Attribute deklariert werden.

Für jedes in K deklarierte Attribut vom Typ T besitzen die Objekte der Klasse K eine ***objektlokale*** Variable vom Typ T. Diese objektlokalen Variablen nennt man häufig auch ***Instanzvariablen***.

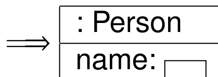
Deklaration von Klassen (4)

Die Lebensdauer der Instanzvariablen ist gleich der Lebensdauer des Objekts.

Klasse: Objekt:



⟹



Deklaration von Klassen (5)

Methoden
Innerhalb der Klassendeklaration können beliebig viele Methoden deklariert werden.

Methodendeklarationen bestehen aus einer Signatur und einem Methodenrumpf. Syntaktisch sind sie wie Prozedurdeklarationen aufgebaut.

Außer den deklarierten Parametern besitzt jede Methode m eine weiteren, sogenannten ***impliziten Parameter*** vom Typ der Klasse, in dem m deklariert wurde. Dieser Parameter wird im Methodenrumpf mit `this` bezeichnet.

Beispiel:

Die obige Methode in Klasse Person:

```
String getName() { return this.name; }
```

hätte in der prozeduralen Programmierung also die Signatur:

```
String getName(Person this)
```

Neben den prozeduralen Anweisungen kann eine Methodenrumpf in Java:

- neue Objekte erzeugen,
- auf Attribute zugreifen,
- Nachrichten an andere Objekte schicken (Methodenaufruf).

Konstruktoren

Konstruktoren erzeugen und initialisieren Objekte.

Sie haben den gleichen Namen wie die Klasse, in der sie deklariert sind.

Beim Start der Ausführung eines Konstruktors ist das zugehörige Objekt bereits erzeugt, seine Attribute jedoch nur mit Standardwerten initialisiert.

Konstruktoren liefern als Ergebnis das neu erzeugte Objekt zurück, genauer eine Referenz auf dieses Objekt.

Unterabschnitt 5.3.2

Anwendung von Objekten

Anwendung von Objekten

Im Allg. umfasst die Anwendung von Objekten vier Anweisungen:

- Objekterzeugung
- Attributzugriff
- Methodenaufruf
- Objektlöschung (in Java nicht direkt unterstützt)

Objekte: Erzeugen und Referenzieren

Syntax in Java:

Objekte werden mit Ausdrücken folgender Form erzeugt (engl. *object creation expression)*:

```
new <Konstruktorname> ( <Parameterliste> )
```

Semantik:

1. Erzeuge ein Objekt / eine Instanz der Klasse, der der Konstruktor gehört. Dabei werden insbesondere die Instanzvariablen angelegt.
2. Werte die aktuellen Parameter aus.
3. Rufe den Konstruktor mit den Parametern auf. Dieser sollte die Instanzvariablen initialisieren.

Ergebnis ist die Referenz des neu erzeugten Objekts.

Begriffsklärung: (Referenz, Verweis, Zeiger)

Eine ***Objektreferenz*** (engl. *object reference*) ist eine eindeutige abstrakte Adresse oder Bezeichnung für ein Objekt. Manchmal spricht man auch von ***Verweis*** (engl. *link*) oder ***Zeiger*** (engl. *pointer*).

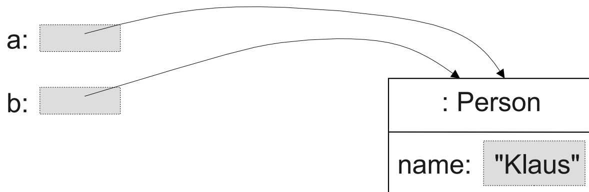
Variablen speichern nicht die Objekte als Ganzes, sondern Objektreferenzen.

Die Auswertung von Ausdrücken eines Klassentyps K liefert Referenzen auf Objekte des Typs K.

Beispiel: (Objektreferenz)

Folgendes Programmfragment verdeutlicht die Unterscheidung zwischen Objekt und Referenzen:

```
Person a,b;
a = new Person ("Klaus");
b = a;      // a und b speichern dieselbe Objektreferenz
b.getName();   // liefert "Klaus"
```



Es gilt also: `a.name==b.name`

Sprechweisen & Bemerkungen

- Häufig spricht man von Referenzen auf ein Objekt.
- Referenzen nennt man auch anonyme Namen oder Bezeichner. Eine Referenz ist ein Wert.
- Referenzen stellt man graphisch üblicherweise durch Pfeile dar. Zwei Referenzen sind gleich, wenn sie auf das gleiche Objekt zeigen.
- Die Unterscheidung Referenz / Objekt hat viele Analogien:
  - Anschrift / Wohnung
  - Email-Adresse / Mailbox
  - Telefonnummer / Telefonanschluss
  - Speicheradresse / Speicherzelle

Beispiel: (Referenzsemantik)

Beispiel zur Unterscheidung zwischen Objekten und Referenzen:

```
class EinObjekt {
   int meinAttribut;
   EinObjekt( int n ) {  meinAttribut = n; }
}
```

Anwendung der Klasse EinObjekt:

```
EinObjekt a, b, c;
a = new EinObjekt(7);
b = a;
c = new EinObjekt(7);
if( a != c ) {  a.meinAttribut = 9; }
System.out.println( b.meinAttribut );
System.out.println( c.meinAttribut );
```

Beispiel: (Referenzsemantik) (2)

Operationen auf Referenzen in Java:

- Referenzen lassen sich mit „==“ auf Gleichheit testen bzw. mit „!=“ auf Ungleichheit. Sie sind genau dann gleich, wenn sie dasselbe Objekt referenzieren.

  Beispiel:
  Nach der Zuweisung an c gilt: a == b und a != c und b != c

- Über Referenzen kann man Objekten Nachrichten schicken und auf sie zugreifen, d.h. auf ihre Instanzvariablen.

Bemerkungen:

- Objekte können der gleichen Klasse angehören und den gleichen Zustand haben (gleich sein), aber trotzdem nicht die gleiche Identität haben und damit auch unterschiedliche Referenzen besitzen.
- Variablen von einem Klassentyp speichern Referenzen. Wir sagen deshalb auch vereinfachend, dass eine Variable ein Objekt referenziert.
- Es ist wichtig zwischen einer Variablen und dem Objekt, dass sie referenziert zu unterscheiden!

Beispiel: (null-Referenz)

Folgendes Programmfragment illustriert die Operationen und Probleme im Zusammenhang mit der null-Referenz:

```
Person a,b;
a = new Person ("Klaus");
b = null;        // zulaessig
if( a != b ) {   // Vergleich ok
  String s;
  s = b.name;    // NullPointerException
  b.getName();   // NullPointerException
}
```

Begriffsklärung: (Objektgeflecht)

Eine Menge von Objekten, die sich gegenseitig referenzieren, nennen wir ein ***Objektgeflecht*** (vgl. Folie **??**).

Bemerkungen:

- Objektgeflechte werden zur Laufzeit aufgebaut und verändert, sind also dynamische Entitäten.
- Klassendiagramme kann man als vereinfachte statische Approximationen von Objektgeflechten verstehen.

Lebensdauer von Objekten und Instanzvariablen:

In Java lassen sich Objekte nicht löschen.

Aus Sicht des Programmierers leben Objekte und deren ***Instanzvariablen*** von der Objekterzeugung bis zum Ende der Ausführung des Programms.

Der Speicher nicht erreichbarer Objekte wird ggf. vor Ablauf der Lebensdauer von der automatischen Speicherbereinigung frei gegeben (vgl. Folien **??**f).

Attributzugriff

Syntax in Java:

Auf Instanzvariablen von Objekten kann mit Ausdrücken folgender Form zugegriffen werden:

<referenzwertiger Ausdruck> . <Attributbezeichner>

Semantik:

Werte den referenzwertigen Ausdruck aus. Liefert dieser null, löse eine NullPointerException aus.

Andernfalls liefert er die Referenz auf ein Objekt X; in dem Fall liefert der gesamte Ausdruck die Instanzvariable von X zum angegenen Attribut (L-Wert) oder deren Wert (R-Wert).

Attributzugriff (2)

Abkürzende Notation:

Der implizite Methodenparameter `this` kann beim Zugriff auf ein Attribut `a` weggelassen werden, d.h.

```
a
```

ist gleichbedeutend mit

```
this.a
```

innerhalb von Klassen, in denen `a` deklariert ist.

Beispiel: (Attributzugriffe)

```
class DeinObjekt {
  MeinObjekt  du;
  String deinName;
}

class MeinObjekt {
   boolean binEingetragen;
   String  meinName;

   void dichInit (DeinObjekt d) {
      System.out.println( d.deinName );
      d.deinName = this.meinName;
      this.binEingetragen = true;
      meinName = d.du.meinName;
   }
}
```

Methodenaufruf: (engl. method invocation)

Syntax:

Ein Methodenaufruf ist ein Ausdruck ähnlich einem Prozeduraufruf, allerdings mit einem zusätzlichen Parameter:

<refwertigerAusdruck> **.** <Methodenbezeichner> (<AktParamliste>)

Methodenaufruf: (engl. method invocation) (2)

Semantik:

Werte den referenzwertigen Ausdruck aus. Liefert dieser null, löse eine NullPointerException aus.

Andernfalls liefert er die Referenz auf ein Objekt X. Werte die aktuellen Parameter $p_1, \ldots, p_n$ aus.

Führe den Rumpf der angegebenen Methode mit

- X als implizitem Parameter und
- $p_1, \ldots, p_n$ als expliziten Parametern aus.

Das Ergebnis des Aufrufs ist der Rückgabewert der Ausführung des entsprechenden Methodenrumpfes.

Bemerkung:

Eine verfeinerte Semantik wird in 5.3 behandelt. Dabei wird auch der Zusammenhang zum Senden von Nachrichten angesprochen.

Abkürzende Notation:

Wie beim Attributzugriff kann auch beim Methodenaufruf der implizite Methodenparameter `this` weggelassen werden, also `m(...)` statt `this.m(...)`.

Beispiel: (Methodenaufrufe)

```
class Mensch {
  Mensch vater , mutter;
  String name;

  Mensch getOpa (boolean mutterseits) {
    if ( mutterseits ) {   return mutter.vater;
    } else {               return vater.vater;
    }
  }

  void eineMethode (Mensch m) {
     Mensch opaV;
     String opaMName;
     opaV = m.getOpa(false);
     opaMName =  m.getOpa(true). name;
} }
```

Objektorientierte Programme

Ein objektorientiertes Java-Programm Π besteht aus einer Menge von Klassen. Mindestens eine der Klassen muss eine Methode mit Namen `main` und folgender Signatur besitzen:

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

Beim Start von Π wird die Klasse angegeben, deren main-Methode ausgeführt werden soll:

```
java <Klassenname> <arg1> <arg2> ...
```

Die Argumente `arg1`,... werden dabei der main-Methode im Parameter `args` als ein Feld von Strings übergeben.

Bei der Ausführung werden die benötigten Objekte erzeugt. Diese Bearbeiten ihre Aufträge durch Ausführung von Methoden.

Unterabschnitt 5.3.3

Spracherweiterungen: Initialisierung und Ausnahmebehandlung

Initialisierung

Attribute und lokale Variablen können direkt an ihrer Deklarationsstelle initialisiert werden.

Somit sind folgende Programmstücke gleichbedeutend.

```
float pi;
pi = 3.141;

class C {
  int a = 78;
  C() {};
}
```

```
float pi = 3.141;

class C {
  int a;
  C() { a = 78;};
}
```

Die Initialisierung von Attributen erfolgt vor dem Eintritt in den Konstruktor*rumpf*.

Initialisierung (2)

In Java können Attribute und Variablen durch das Schlüsselwort `final` als ***unveränderlich*** deklariert werden.

In diesem Fall muss die Initialisierung an der Deklarationsstelle erfolgen oder in geeigneter Weise im Konstruktor.

```
class Mathe {
    ...
    final float pi = 3.141; // Konstante
    ...
}
```

Beispiel: (Initialisieren von Attributen)

Folgende Programmstücke sind gleichbedeutend:

```
class C {
  int ax = 7, ay = 9;

  C(){
    m();
  }

  void m(){
    int v1 = 4848, v2 = -3;
    ...
  }
}
```

```
class C {
  int ax, ay;

  C(){
    ax = 7;
    ay = 9;
    m();
  }

  void m(){
    int v1, v2;
    v1 = 4848;
    v2 = -3;
    ...
  }
}
```

Ausnahmebehandlung

Wie in 3.1., Folie 187, bereits angesprochen, kann die Auswertung eines Ausdrucks bzw. die Ausführung einer Anweisung:

- normal terminieren
- in eine Ausnahmesituation kommen und abrupt terminieren
- nicht terminieren

Ausnahmebehandlung (2)

Es gibt drei Arten von Ausnahmesituationen:

1. Vom Programmierer schwer zu kontrollierende und zu beseitigende Situtationen (z.B. Speichermangel)
2. Programmierfehler (z.B. Nulldereferenzierung, Verletzung von Indexgrenzen)
3. Zeitweise nicht verfügbare Ressourcen, anwendungsspezifische Ausnahmen, die behebbar sind.

Ausnahmen werden in Programmiersprachen verschieden behandelt:

- Programmabbruch (engl. abortion)
- Ausnahmebehandlung

Ausnahmebehandlung (3)

Java bietet Sprachmittel für die ***Ausnahmebehandlung*** (engl. ***exception handling***).

Dabei spielen drei Aspekte eine Rolle:

1. Wann/wie werden Ausnahmen ausgelöst?
2. Wie kann man sie abfangen?
3. Wie kann man neue Ausnahmetypen deklarieren?

Auslösen von Ausnahmen:

Das Auslösen einer Ausnahme kann

- sprachdefiniert (z.B. NullPointer, IndexOutOfBounds)
- oder durch eine Anweisung spezifiziert sein.

In Java gibt es zum Auslösen von Ausnahmen die throw-Anweisung.

throw-Anweisung

Syntax:
Die throw-Anweisung hat die Form:

```
throw <Ausdruck>;
```

wobei der Ausdruck ein Ausnahmeobjekt als Ergebnis liefern muss.

Semantik:
Werte den Ausdruck aus.

Löst die Auswertung eine Ausnahme aus, ist dies die Ausnahme, die von der Anweisung ausgelöst wird.

Andernfalls löse die Ausnahme aus, die das Ergebnis des Ausdrucks ist.

Abfangen von Ausnahmen:

Die try-catch-Anweisung dient dem Abfangen und Behandeln von Ausnahmen:

```
void myMethod (String[] sfeld)
{
    try {
      myPrint( sfeld[0] );
      myPrint( sfeld[1] );
  } catch (NullPointerException e) {
      myPrintln("sfeld is null");
  } catch (IndexOutOfBoundsException e){
      myPrintln("sfeld too small");
  }
}
```

Abfangen von Ausnahmen: (2)

Tritt eine Ausnahme vom Typ A im try-Block auf, wird ein A-Objekt X erzeugt. Ist der Typ A in der Liste der catch-Klauseln aufgeführt,

- wird die Ausnahme *gefangen* (exception is catched),
- X an den Bezeichner der entsprechenden catch-Klausel gebunden und
- diese catch-Klausel ausgeführt (Verfeinerung in 5.3)

Benutzerdefinierte Ausnahmetypen:

Die Deklaration von Exception-Klassen behandeln wir in Abschnitt 5.3.

Bemerkung:
Java verlangt die Deklaration bestimmter Ausnahmetypen in der Signatur einer Methoden m, wenn sie nicht von m abgefangen werden (Genaueres in 5.3).

Beispiel:

```
int m( int i ) throws SomeException {
    if( i<0 ) {
       throw new SomeException();
    }
    ...
}
```

Beispiel: (Ausnahmebehandlung)

```
public static void main( String[] argf ){
 long maxint = 2147483647L;
 try{
     int m, n, ergebnis = 0 ;
     m = Integer.parseInt( argf[0] );
     n = Integer.parseInt( argf[1] );
     long aux = (long)m + (long)n;
     if( aux > maxint )   throw new Ueberlauf();
     ergebnis =  (int)aux ;
 } catch ( IndexOutOfBoundsException e ) {
     System.out.println("Falsche Argumente");
 } catch ( NumberFormatException e ) {
     System.out.println("Parameter keine int-Konstante");
 } catch ( Ueberlauf e ) {
     System.out.println("Ueberlauf");
}}
```

Unterabschnitt 5.3.4

Anwenden und Entwerfen von Klassen

Anwenden und Entwerfen von Klassen

Klassen bilden das zentrale Sprachkonstrukt von Java.

Dementsprechend stehen beim Programmentwurf zwei Fragen im Mittelpunkt:

- Welche existierenden Klassen können für den Programmentwurf herangezogen werden?
- Welche Klassen müssen neu entworfen werden?

Zur Diskussion dieser Aspekte betrachten wir ein kleines Beispiel.

Aufgabenstellung:

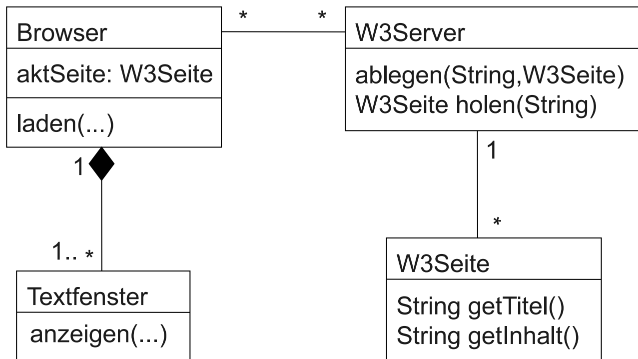
Ein rudimentäres Browser-Programm soll realisiert werden, mit dem einfache W3Seiten bei einem Server geholt und in einem Fenster angezeigt werden können.

Wir gehen davon aus, dass die folgenden Klassen existieren:

- `W3Seite`: Implementiert W3Seiten.
- `W3Server`: Implementiert W3Server bzw. ihre Schnittstelle.
- `Textfenster`: Kann W3-Seiten anzeigen.

Aufgabenstellung: (2)

Klassendiagramm zur Lösung der Aufgabenstellung:



Aufgabenstellung: (3)

Schnittstellen der gegebenen Klassen:

```
class W3Server {
  W3Server() { ... }
  void ablegenSeite( String adr, W3Seite s ){
    ...
  }
  W3Seite holenSeite( String adr ) { ... }
}

class TextFenster ... {
  ...
  TextFenster() { ... }
  void anzeigen(String tzeile,String text){
     ...
  }
}
```

Realisierung:

Die vollständige Klasse W3Seite:

```
/** Objekte repraesentieren triviale
 *  Web-Seiten mit Titelzeile und Inhalt
 */
class W3Seite {
   String titel, inhalt;

   W3Seite ( String t, String i ) {
      titel  = t;
      this.inhalt = i;
   }

   String getTitel() { return this.titel; }

   String getInhalt(){ return inhalt; }
}
```

Realisierung: (2)

Wichtige Implementierungsteile einer rudimentären Browser-Klasse:

```
class Browser {
  W3Server     meinServer;
  TextFenster  oberfl;
  W3Seite      aktSeite;   // aktuelle Seite

  Browser( W3Server server ){
    meinServer = server;
    oberfl     = new TextFenster();
    laden( new W3Seite("Startseite",
           "NetzSurfer: Keiner ist kleiner") );
    interaktiveSteuerung();
  }
```

Realisierung: (3)

```
  void laden( W3Seite s ){
    aktSeite = s;
    oberfl.anzeigen( aktSeite.getTitel(),
                     aktSeite.getInhalt());
  }
  void interaktiveSteuerung() { ... }
}
```

Unterabschnitt 5.3.5

Spracherweiterungen: Überladen, Klassenvariablen und -methoden

Überladen

In Java ist es erlaubt, innerhalb einer Klasse mehrere Methoden mit dem gleichen Namen zu deklarieren, d.h. es gibt zwei Bindungen mit gleichem Namen.

Eine derartige Mehrfachverwendung nennt man Überladen eines Namens. Methoden mit gleichen Namen müssen sich in der Anzahl oder in den Typen der Parameter unterscheiden.

Durch die unterschiedliche Signatur kann der Übersetzer die Überladung auflösen, d.h. für jede Aufrufstelle ermitteln, welche von den Methoden gleichen Namens an der Aufrufstelle gemeint ist.

Entsprechend dem Überladen von Methodennamen erlaubt Java auch das Überladen bei Konstruktoren.

Beispiel: (Überladen)

Die Java-Bibliothek bietet viele Beispiele für Überladung. Wir betrachten die Klasse String (hier nur unvollständig wiedergegeben):

```
class String {
 char[] value; /* value is used for character storage */
 /** The offset is the first index of the used storage*/
 int offset;
 /** The count is the number of characters in the ... */
 int count;

 String() { value = new char[0]; }
 String( String value ) { ... }
 String( char[] value ) {
   this.count = value.length;
   this.value = new char[count];
   System.arraycopy(value,0,this.value,0,count);
 }
```

Beispiel: (Überladen) (2)

```
  ...
  int indexOf(int ch) { return indexOf(ch, 0);}
  int indexOf(int ch, int fromIndex) { ... }
  int indexOf(String str) { ...}
  int indexOf(String str, int fromIndex) {...}
  int length() { return count; }
  char charAt(int index) {
    if ((index < 0) || (index >= count)) {
     throw
      new StringIndexOutOfBoundsException(index);
    }
    return value[index + offset];
  }
}
```

Klassenattribute

Die Deklaration eines ***Klassenattributs*** liefert eine klassenlokale Variable. Klassenattribute/-variablen werden häufig auch als ***statische Attribute/Variablen*** bezeichnet.

Syntax:

```
static <Typausdruck> <Attributname> ;
```

Die Variable kann innerhalb der Klasse mit dem Attributnamen, außerhalb mittels

```
<Klassenname> . <Attributname>
```

angesprochen werden.

Klassenattribute (2)

Die Lebensdauer der Variablen entspricht der Lebensdauer der Klasse.

Bemerkung:

Klassenvariablen verhalten sich ähnlich wie globale Variablen in der prozeduralen Programmierung.

Beispiel: (Klassenattribut)

```
class InstanceCount {
  static int instCount = 0;
  InstanceCount(){
    instCount++;
    ...
  }  ...
}
```

Klassenmethode

Die Deklaration einer ***Klassenmethode*** entspricht der Deklaration einer Prozedur. Sie werden häufig auch als ***statische Methoden*** bezeichnet.

Klassenmethoden besitzen keinen impliziten Parameter.

Sie können nur auf Klassenattribute, Parameter und lokale Variable zugreifen.

Syntax:

```
static <Methodendeklaration>
```

Klassenmethoden werden mit folgender Syntax aufgerufen:

```
<Klassenname> . <Methodenname> ( ... )
```

Innerhalb der Klasse kann der Klassenname entfallen.

Beispiel: (Klassen-, statische Methoden)

Deklaration:

```
class String {
    ...
    static String valueOf( long l )  { ... }
    static String valueOf( float f ) { ... }
    ...
}
```

Anwendung/Aufruf:

```
    String.valueOf( (float)(7./9.) )
```

liefert die Zeichenreihe: "0.7777778"

Bemerkung:

In Kapitel 4 wurden Klassenmethoden zur prozeduralen Programmierung in Java genutzt.

Beispiele: (Klassenattribute u. -methoden)

1. Charakteristische Beispiele für Klassenattribute und -methoden liefert die Klasse `System`, die eine Schnittstelle von Programmen zur Umgebung bereitstellt:

```
class System {
  final static InputStream in  = ...;
  final static PrintStream out = ...;
  static void exit(int status) { ... }
  static native void arraycopy(
    Object src,int src_position,
    Object dst,int dst_position, int length);
}
```

Beispiele: (Klassenattribute u. -methoden) (2)

Die Klasse PrintStream besitzt Methoden print und println:

```
System.out.print("Das klaert die Syntax");
System.out.println(" von Printaufrufen");
```

Beispiele: (Klassenattribute u. -methoden) (3)

2. Statische Methoden und Überladung in Klasse IO:

```
public class IO {
  public static void print(Object object) {
    System.out.print(object);
  }
  public static void println(Object object) {
    System.out.println(object);
  }

  public static int readInt() {
    try {
      return new Scanner(System.in).nextInt();
    } catch(Throwable e) {
      throw new RuntimeException("Not an integer",e);
    }
  }
```

Beispiele: (Klassenattribute u. -methoden) (4)

```
    public static String readString() {
      try {
        return new Scanner(System.in).nextLine();
      } catch(Throwable e) {
        throw new RuntimeException("Error in input",e);
    } }
    public static char readChar() {
      try {
        String result = readString();
        if( result.length() > 1 ) {
           System.out.println("*** Input too long.");
        }
        return result.charAt(0);
      } catch(IndexOutOfBoundsException ex) {
        throw new RuntimeException("Empty input",ex);
} } }
```

Unterabschnitt 5.3.6

Zusammenwirken der Spracherweiterungen

Zusammenwirken der Spracherweiterungen

Das Zusammenwirken der eingeführten Sprachelemente erlaubt bereits, recht komplexe Programme zu schreiben.

Folgendes Programmbeispiel mischt prozedurale und objektorientierte Sprachelemente. Es dient zum Studium des Zusammenwirkens der Spracherweiterungen.

Beispiel: (Zusammenwirken von Sprachel.)

Wir erweitern das Browserbeispiel von 5.2.4:

- Unterstützung mehrerer Browserobjekte
- Interaktive Steuerung über die Konsole

```
class Konsole {
  static String readString() { ... }
  static void writeString( String s ) {...}
}
```

Beispiel: (Zusammenwirken von Sprachel.) (2)

Entwurf der Implementierung:

- Die gemeinsamen Teile aller Browserfenster werden durch Klassenattribute und –methoden realisiert.
- Es gibt zwei Konstruktoren: Einer startet das erste Browserobjekt; der andere weitere Browserobjekte.
- Die gemeinsamen Teile der Konstruktoren werden von der Methode `initialisieren` erledigt.

Beispiel: (Zusammenwirken von Sprachel.) (3)

- Die interaktive Steuerung von der Konsole wird durch eine statische Methode implementiert.
- Zur einfacheren Handhabung steht eine Klassenmethode `start` zur Verfügung, die den W3Server als Argument bekommt:

```
...
Browser.start( testServer );
...
```

- Die Browserfenster werden in einem Feld auf Klassenebene verwaltet.

Beispiel: (Zusammenwirken von Sprachel.) (4)

```
class Browser {
  TextFenster oberfl;
  W3Seite     aktSeite;
  static W3Server  meinServer;
  static final int MAX_ANZAHL = 4;
  static Browser[] gestarteteBrowser =
                     new Browser[MAX_ANZAHL];
  static int       naechsterFreierIndex = 0;
  static int       aktBrowserIndex;
  static W3Seite   startseite =
   new W3Seite("Startseite",
           "NetzSurfer: Keiner ist kleiner");
```

Beispiel: (Zusammenwirken von Sprachel.) (5)

```
// Konstruktor fuer ersten Browsers
Browser( W3Server server ) {
  if( naechsterFreierIndex != 0 ) {
    System.out.println("Browser gestartet");
  } else {
    meinServer = server;
    initialisieren();
  }
}
```

Beispiel: (Zusammenwirken von Sprachel.) (6)

```
// Konstruktor fuer weiterere Browserfenster
Browser() {
  if( naechsterFreierIndex == MAX_ANZAHL ) {
    System.out.print("Maximale Anzahl ");
    System.out.println(" Browser erreicht");
  } else {
    initialisieren();
  }
}
```

Beispiel: (Zusammenwirken von Sprachel.) (7)

```
static void start( W3Server server ) {
  new Browser(server);
  Browser.interaktiveSteuerung();
}

void initialisieren() {
  oberfl           =  new TextFenster();
  gestarteteBrowser[naechsterFreierIndex] = this;
  aktBrowserIndex =  naechsterFreierIndex;
  naechsterFreierIndex++ ;
  laden( startseite );
}
```

Beispiel: (Zusammenwirken von Sprachel.) (8)

```
void laden( W3Seite s ) {
  aktSeite = s;
  oberfl.anzeigen( aktSeite.getTitel(),
                   aktSeite.getInhalt() );
}
```

Beispiel: (Zusammenwirken von Sprachel.) (9)

```
static void interaktiveSteuerung() {
  char steuerzeichen = '0';
  do {
    Konsole.writeString("Steuerzeichen [lnwe]: ");
    try {
      String eingabe = Konsole.readString();
      if( eingabe.equals("") )
        steuerzeichen = '0';
      else
        steuerzeichen = eingabe.charAt(0);
    } catch( Exception e ) {
      System.exit( 0 );
    }
    switch( steuerzeichen ){
    case 'n':  // neues Browserfenster
      new Browser();
      break;
    case 'l':  // Laden einer Seite
      Konsole.writeString("Seitenadresse: ");
      String seitenadr = Konsole.readString();
      gestarteteBrowser[aktBrowserIndex] .
```

Beispiel: (Zusammenwirken von Sprachel.) (10)

```
        laden( meinServer.holenSeite( seitenadr ) );
      break;
    case 'w':  // Wechsel des Fokusses
      aktBrowserIndex =
        (aktBrowserIndex+1) % naechsterFreierIndex;
      break;
    case 'e': System.exit(0); //Beenden des Programms
    default:
      Konsole.writeString("falsche Eingabe\n");
    }
  } while( true );
} } // Ende der Klasse Browser
```