PD Dr.-Ing. habil. Stefan Schwarz

Einführung in die Objektorientierte Programmierung mit *Java*

Erstellt: 06.05.99 12:04

Inhalt

1 Allgemeines 7

1.1 WAS IST JAVA 7

1.2 ANWENDUNGSGEBIETE 9

2 Einführung 11

2.1 EIN ERSTES BEISPIEL 11

2.1.1 Java-Applet 12

2.1.1.1 Java-Source 13

2.1.1.2 HTML-Source 13

2.1.2 Java-Applikation 14

2.1.2.1 Java-Source 15

2.1.3 Typische Bestandteile eines Java-Programms 15

3 Java-Grundlagen 17

3.1 JAVA-GRAMMATIK 17

3.1.1 Kommentar 17

3.1.2 Namen (Variable, Funktionen) 17

3.1.3 Variable 17

3.2 OPERATOREN 19

3.2.1 Vorrang der Operatoren 22

3.2.2 Zuweisungsoperator 22

3.3 SCHLEIFEN 23

3.3.1 for-Schleife 23

3.3.2 while-Schleife 24

3.3.3 do-while-Schleife 24

3.3.4 Spezielle Schleifenanweisungen 25

3.4 KONTROLLSTRUKTUREN (ABLAUFKONTROLLE) 25

3.4.1 Bedingte Anweisung (if) 26

3.5 SCHLÜSSELWÖRTER 27

3.6 ESCAPE-SEQUENZEN (IN ZEICHENKETTEN) 27

3.7 UNTERSCHIEDE ZU C,C++ 28

3.7.1 Fehlende Features 28

3.7.2 Neue Features 29

3.8 PACKAGES 30

3.8.1 Vorhandene Packages 32

3.8.2 Beispiel zur Nutzung von Klassen verschiedener Packages 34

3.9 AUSNAHMEBEHANDLUNG (EXCEPTIONS) 36

3.10 JAVA-SICHERHEITSKONZEPT 39

4 Konzepte der objektorientierten Programmierung (OOP) 41

4.1 BEGRIFFE DER OOP 41

4.2 (MEMBER-)FUNKTIONEN ... 43
4.3 KLASSENKONZEPT (VERERBUNG) ... 45
4.3.1 Basisklasse (Superclass) ... *45*
4.3.2 Abgeleitete Klasse (Subclass) ... *45*
4.3.3 Abstrakte Klasse (abstract class) ... *45*
4.3.4 Mehrfachvererbung (multiple inheritance) ... *46*
4.3.5 Schnittstellen (interfaces) ... *47*
4.3.6 Implementierung von Funktionen ... *47*
4.3.7 Statische Memberfunktionen ... *48*
4.4 NOTATION ... 49
4.5 KLASSENDESIGN ... 49
4.6 REALISIERUNG IN JAVA ... 50
4.6.1 Klassendefinition ... *51*
4.6.2 Methodendefinition ... *51*
4.7 BEISPIEL ... 52
4.7.1 Design ... *52*
4.7.1.1 Grobentwurf ... 52
4.7.1.2 Feinentwurf ... 53
4.7.1.3 Implementierung ... 54

5 Benutzeroberflächen ... 61
5.1 ABSTRACT WINDOW TOOLKIT (AWT) ... 61
5.1.1 Objekthierarchie vs. Klassenhierarchie ... *62*
5.1.2 Objekte erzeugen ... *63*
5.1.3 Benutzerinteraktion ... *63*
5.1.4 Layout ... *66*
5.1.4.1 Layoutbeispiel ... 67

6 Parallele Prozesse (Threads) ... 73
6.1 ALLGEMEINES ... 73
6.2 IMPLEMENTIERUNG VON THREADS ... 73
6.3 IMPLEMENTIERUNG ÜBER KLASSE „THREAD“ ... 75
6.4 IMPLEMENTIERUNG ÜBER SCHNITTSTELLE „RUNNABLE“ ... 76
6.5 IMPLEMENTIERUNG DYNAMISCHER OBJEKTE ... 77
6.6 SYNCHRONISATION ZWISCHEN THREADS ... 79

7 Datenbankzugriff ... 83
7.1 SINNVOLLE KLASSENHIERARCHIE ... 84
7.1.1 Klasse "Database" ... *86*
7.1.2 Klasse "GrafikDatabase" ... *87*

8 Verteilte Anwendungen ... 95
8.1 BESTANDTEILE EINER VERTEILTEN JAVA-ANWENDUNG ... 95

8.2 REMOTE METHOD INTERFACE (RMI) 96
8.2.1 Schnittstelle (Interface) *97*
8.2.2 Serverkomponente *97*
8.2.3 Clientkomponente *99*
8.2.4 Kommunikation *101*
8.3 ZUSAMMENFASSUNG 103

9 Anhang 105

9.1 OBJECT MODELING TECHNIQUE (OMT) 105
9.2 KURZBESCHREIBUNG SQL-SYNTAX 107
9.2.1 Allgemeine Syntax *107*
9.2.2 Select-Anweisung *107*
9.2.3 Insert-Anweisung *108*
9.2.4 DELETE-Anweisung *108*
9.2.5 Update-Anweisung *108*
9.3 EINRICHTEN EINER ODBC-DATENBANK UNTER WINDOWS 109
9.4 ARBEITEN MIT VISUAL J++ 115
9.4.1 Grundlagen *115*
9.4.2 Arbeiten mit mehreren Projekten *118*

1 Allgemeines

1.1 Was ist Java

Java ist eine neue, universell einsetzbare, objektorientierte Programmiersprache. Sie wurde von der Fa. Sun Microsystems entwickelt (Beginn 1991, Projekt „Green“), um über unterschiedlichste Rechnerplattformen hinweg ein einziges Programm unverändert lauffähig zu haben. Dies ist natürlich dann besonders sinnvoll, wenn verschiedene Rechnerarchitekturen in einem Netzwerk miteinander verbunden sind und alle diese Rechner ein gemeinsames Programm, welches an zentraler Stelle vorgehalten wird, nutzen sollen. Dieses Konzept des „Write once, run anywhere“ hat sich daher besonders mit der Verbreitung des Internets durchgesetzt und daher ist Java im Bereich des Internets bzw. Intranets zu der Standardprogrammiersprache geworden.

Der Name „Java“ kommt übrigens von der gleichnamigen indonesischen Insel. Er entstammt einfach aus der Tatsache, daß während der Entwicklung zu Java Unmengen Kaffee verkonsumiert wurden, und der Begriff „Java“ steht im Amerikanischen umgangssprachlich für Kaffee. Daher auch das Java-Logo

Die Umsetzung des Konzepts zeigt Abbildung 1-1. Der Unterschied zu herkömmlichen Programmiersprachen wie C oder C++ liegt einfach darin, daß der Java-Quellcode nicht in einen maschinenabhängigen, sondern in einen maschinenunabhängigen Code (sog. Bytecode, in Abbildung 1-1 mit „Java-Software“ bezeichnet) übersetzt wird. Dieser Bytecode wird dann zur Laufzeit durch die „Java Virtual Machine“ (JVM) in einen maschinenabhängigen Code übersetzt und ausgeführt. Da diese Übersetzung zur Laufzeit natürlich einen zusätzlichen Aufwand erfordert wird zunächst einmal ein Java-Programm langsamer ablaufen als herkömmliche übersetzte Programme. Allerdings wird dies meistens dadurch wieder ausgeglichen, daß bei moderneren Systemen ein Compiler mit in die virtuelle Maschine integriert ist (JIT = „Just in time“ compiler), welcher die Software beim erstmaligen Laden kompiliert und somit im Laufzeitverhalten deutlich beschleunigt.

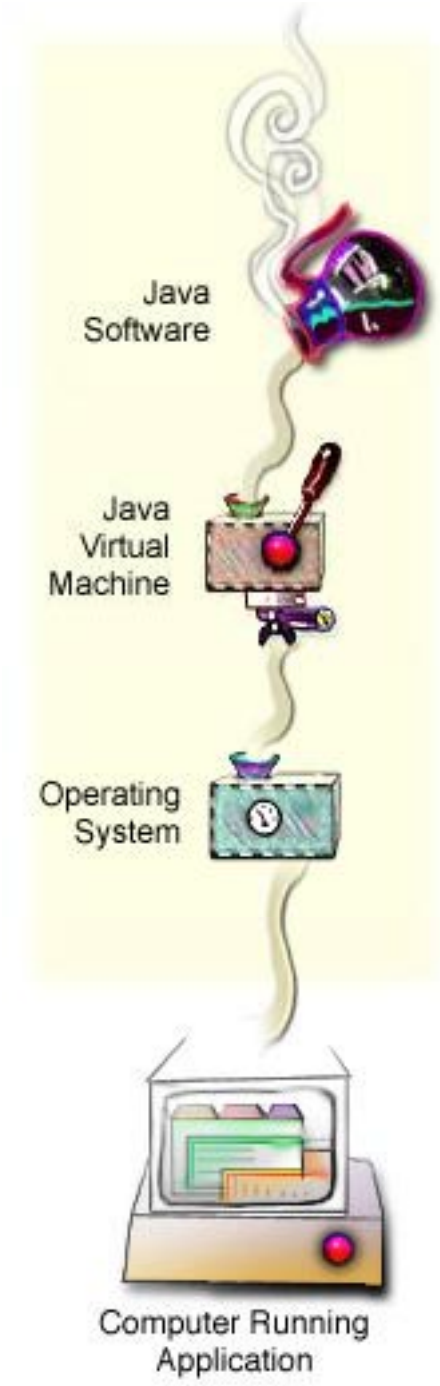


Abbildung 1-1 Ausführung von Java-Software

Der große Vorteil dieser Architektur ist nun, daß Java-Programme (im Bytecode, nicht Quellcode) irgendwo in einem Netzwerk bereitgestellt werden können und beliebige Rechner diese Programme über das Netz laden und ausführen können. Damit entfällt auch der bei compilierten Sprache notwendige Schritt des Bindens (Linking), da in Java weitere Module während der Laufzeit einfach neu hinzugeladen werden können. Ein Java-Programm kann also auch während der Laufzeit dynamisch verändert werden. Prinzipiell ist es sogar möglich,

daß ein Java-Programm sich selbst dadurch erweitert, daß es neue Module zur Laufzeit generiert.

Die Implementierung der „Java Virtual Machine“ kann auf 2 Arten erfolgen. Einmal wird die JVM in die gewohnten Internetbrowser (Netscapes Communicator, Microsofts Internet Explorer) integriert. Damit können Java-Programme mit dem Browser geladen und innerhalb des Browsers auch ausgeführt werden. Es wird keine zusätzliche Software benötigt. Solche Java-Programme werden als „Applets“ bezeichnet. Die 2. (wenn auch weit weniger eingesetzte) Möglichkeit besteht darin, die JVM als getrennte Software auf einem Rechner zu installieren und das entsprechende Java-Programm über die JVM zu laden und auszuführen. Für die gängigen Plattformen ist diese Software über das Internet frei verfügbar, allerdings ist das Starten einer Java-Applikation auf diesem Wege etwas mühsamer. Diese Java-Programme bzeichnet man als „Applikationen“. Im folgenden Kapitel werden diese beiden Wege noch näher beleuchtet.

1.2 Anwendungsgebiete

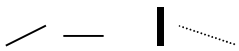
Aufgrund der Plattformunabhängigkeit ist Java natürlich prädestiniert für verteilte Anwendungen in Netzwerken (wie z.B. Internet). Allerdings ist Java eine allgemein verwendbare Sprache für praktisch alle Anwendungsgebiete. Hinzu kommt noch, daß es bei Java im Gegensatz zu praktisch allen anderen Programmiersprachen eine Vielzahl von einheitlichen Programmierschnittstellen gibt, wie z.B. zur Realisierung von

- Benutzeroberflächen
- 2D und 3D-Grafiken
- verteilten Prozessen
- Prozeßkommunikation
- Datenbankanbindungen

Damit bietet Java dem Softwareentwickler ein reichhaltiges Angebot von Funktionalitäten, welche bei anderen Programmiersprachen erst zusätzlich (plattformabhängig) erworben werden müssen.

2 Einführung

Zum besseren Verständnis des folgenden Beispiels ist hier ein kleiner Vorgriff auf die Grundbegriffe der objektorientierten Programmierung (vgl. 4) erforderlich. Zunächst sind die Begriffsdefinitionen für "Klasse" und "Objekt" ausreichend.

Klasse	**Objekt**
entspricht einem Datentyp, d.h. es ist zunächst nur eine abstrakte Definition, also die Möglichkeit, aus diesem Datentyp reale Objekte zu erzeugen. Beispiel: Klasse Linie Eine Klasse beschreibt nur Eigenschaften (z.B. Aussehen, Verhaltensweisen etc.). Diese Eigenschaften werden über Variable und Funktionen abgebildet.	Ein Objekt ist immer etwas real existierendes auf der Basis einer Klasse. Objekte werden immer aus Klassen erzeugt, d.h. zu einer Klasse kann es beliebig viele Objekte geben. Jedes Objekt ist für sich einzigartig. Beispiele:  Alle Objekte in obigem Beispiel besitzen den gleichen Typ (Klasse "Linie"), aber sie sind dennoch einzigartig und voneinander unabhängig. Ein Objekt spiegelt immer das wieder, was innerhalb seiner Klasse an Eigenschaften vordefiniert wurde.

2.1 Ein erstes Beispiel

Es wird eine einfache Java-Anwendung sowohl als Applet als auch als eigenständige Applikation entwickelt, bei der innerhalb eines Grafikfensters einfache Linien dargestellt werden können. Es wird dabei schon eine vollständig objektorientierte Vorgehensweise verfolgt, indem zunächst Objekte des Typs "Linie" und dann die dieses Objekte benutzende Anwendung erstellt werden. Die Klasse "Linie" wird in der Datei "Linie.java" wie folgt implementiert:

```
import java.awt.*;

//Definition einer Linie
public class Linie
{

   //Konstruktor
   public Linie( int x1, int y1, int x2, int y2)
   {
      this.x1 = x1;
      this.x2 = x2;
      this.y1 = y1;
      this.y2 = y2;
   }

   //Methoden (Zeichnen der Linie)
   public void zeichnen(Graphics g)
   {
      g.drawLine (x1, y1, x2, y2);
   }


   //Variablen (Anfangs/Endpunkt einer Linie)
   private int x1 , y1 , x2 , y2;
}
```

Jede Linie besitzt einen aktuellen Status, hier ist es die Kennzeichnung der Anfangs- und Endkoordinate. Dieser Status wird über einfache Variable (x1, y1 etc.) abgebildet. Außerdem besitzt jede Linie die Fähigkeit zur grafischen Darstellung innerhalb eines Objekts des Typs "Graphics", welches von Java zur grafischen Darstellung innerhalb eines Fensters vorgegeben wird. Fähigkeiten von Objekten werden über Funktionen (zeichnen) angesprochen.

2.1.1 Java-Applet

Das Applet definiert einige Linienobjekte und veranlaßt die grafische Darstellung innerhalb des Browser-Fensters. Es definiert sich als Erweiterung der in Java bereits vorhandenen Klasse "Applet" innerhalb der Datei "Hello.java". Die Notwendigkeit der grafischen Darstellung (Update des Fensters nach Vergrößerung, Überdeckung etc.) wird von Java selbst erledigt, indem bei einem erforderlichen Neuzeichnen die Funktion "paint" automatisch aufgerufen wird.

2.1.1.1 Java-Source

```
import java.applet.*;
import java.awt.*;

//Klassen-Deklaration eines Applets
public class Hello extends Applet
{
   //Konstruktor, erzeugt ein Object der Klasse "Hello"
   public Hello()
   {
      l1 = new Linie (0, 0, 100, 100);
      l2 = new Linie (100, 100, 200, 100);
   }

   //Zeichnen des Objekts
   public void paint(Graphics  g)
   {
      l1.zeichnen(g);
      l2.zeichnen(g);
   }

   //Variable
   Linie l1 , l2;
}
```

Nach dem Start des Applets (vgl. 2.1.1.2) wird automatisch der zugehörige Konstruktor für ein Objekt der Klasse "Hello" aufgerufen. Der weitere Programmablauf ergibt sich dann aus der Benutzerinteraktion.

2.1.1.2 HTML-Source

Ein Applet benötigt stets einen WWW-Browser zur Darstellung (vgl. 1.1). Der Browser lädt nicht direkt den Java-Code, sondern eine zugehörige HTML-Seite mit einem entsprechenden Verweis (Applet-Tag) auf die Klasse, welche als erste instanziiert werden soll, d.h. welches Objekt soll als erstes Objekt erzeugt werden (in unserem Beispiel "Hello").

```
<html>
<head>
<title>Hello</title>
</head>
<body>
<hr>

<applet
   code=Hello
   width  = 400
   height = 200>
</applet>

<hr>
</body>
</html>
```

2.1.2 Java-Applikation

Eine Applikation unter Java besitzt zunächst kein Fenster (im Gegensatz zum Applet, welches das Fenster des WWW-Browsers nutzt) zur Darstellung (auf rein textorientierte Applikationen soll nicht näher eingegangen werden). Daher muß die Applikation zunächst ein erstes Fenster (des Typs „Frame“) vereinbaren. Anschließend läuft die Applikation ganz analog zum Applet. Die vorher beim Applet definierte Klasse "Linie" wird hier mit benutzt.

2.1.2.1 Java-Source

```
import java.awt.*;

//Klassen-Deklaration einer Applikation
public class HelloApp extends Frame
{
   //Konstruktor, erzeugt ein Object der Klasse "Hello"
   public HelloApp()
   {
      l1 = new Linie (0, 0, 100, 100);
      l2 = new Linie (100, 100, 200, 100);

      //Erzeugen des Frame-Windows
      setTitle("Hello-Applikation");
      resize(300,200);
      show();
   }

   //Zeichnen des Objekts
   public void paint(Graphics  g)
   {
      l1.zeichnen(g);
      l2.zeichnen(g);
   }

   public static void main (String args[])
   {
      new HelloApp();
   }

   //Variable
   Linie l1 , l2;
}
```

2.1.3 Typische Bestandteile eines Java-Programms

Java-Kennwort	**Bedeutung**
import	Importieren von zusätzlicher Java-Funktionalität. Es handelt sich dabei um ein Java-package (siehe 3.8), welches innerhalb dieser Datei verfügbar gemacht wirtd. Alles was über Standardeigenschaften von Java (dem Package java.lang) hinausgeht muß auf diese Art verfügbar gemacht werden(Ähnlich zum #include bei C++)
class	Definition einer Klasse. Es muß mindestens eine öffentlich zugängliche (public) Klasse vorhanden sein. Der Name der Datei (*name*.java) entspricht dabei dem öffentlich zugänglichen Klassennamen.
paint	Diese Methode wird immer (automatisch durch JVM) aufgerufen, wenn der Fensterinhalt erneuert werden muß. Durch Definition dieser

	Methode in den eigenen Klassen wird damit der grafische Fensterinhalt angegeben.
main	Jede Java-Applikation (nicht Applet!) benötigt eine main-Funktion (analog C++). Diese muß in der angegebenen Form definiert werden.
new	Neue Objekte werden immer mit dem new-Operator definiert. (Nicht zu verwechseln mit dem new-Operator in C++ zur Speicherverwaltung)

3 Java-Grundlagen

In diesem Kapitel werden die Grundelemente der Sprache Java zusammengestellt. Diese Elemente sind eng an C bzw. C++ angelehnt und somit können somit aus diesen Sprachen weitgehend übernommen werden.

3.1 Java-Grammatik

3.1.1 Kommentar

In das Programm eingestreute Kommentare verbessern die Lesbarkeit und Nachvollziehbarkeit.

Syntax:

```
// einzeiliger Kommentar bis Zeilenende
/* mehrzeiliger
   Kommentar
   (keine Verschachtelungen möglich!!)
*/
```

3.1.2 Namen (Variable, Funktionen)

Obwohl Java einen 16-bit Zeichensatz (Unicode) benutzt und damit auch Umlaute darstellen kann, haben einige Implementierungen noch Probleme bei der Verwendung von Umlauten. Daher sollten Umlaute bei der Namensgebung nicht verwendet werden. Dies sollte auch für Dateinamen gelten.

Erlaubt sind beliebig lange Kombinationen aus Buchstaben (a-z,A-Z), Unterstrich (_) und Ziffern (0-9, allerdings nicht als erstes Zeichen eines Namens).

Beispiele:

```
zeile1, zeile_2, gesamteAnzahlAllerPunkte
```

Es wird zwischen Groß- und Kleinschreibung unterschieden. Üblichweise beginnen Namen mit Kleinbuchstaben.

3.1.3 Variable

Variable nehmen zur Laufzeit eines Programms unterschiedliche (variable) Werte an. Sie dienen also dazu, verschiedene Zustände des Programms zu speichern und mit diesen

Werten zu arbeiten. Variable müssen vor ihrer erstmaligen Verwendung definiert werden durch:

Datentyp *name[n]*

Zulässige Datentypen sind die Standarddatentypen von Java (siehe Tabelle 1) bzw. Java-Klassen. Die Namensgebung erfolgt nach den Regeln unter 3.1.2.

Beispiele

```
//Standardtypen
int anzahl;
long wert1, wert2, wert3;
float x, y=1.5, z=3.0;

//Objekte (aus Klassendeklarationen)
Linie l1, l2 = new Linie(0,0,100,100);
```

Alle Variable werden in Java mit einem Standardwert vorbelegt (initialisiert), sofern keine spzielle Wertzuweisung während der Definition erfolgt. Der Standardwert für nicht initialisierte Objekte (im obigen Beispiel „l1“) ist immer die vordefinmierte Konstane „null“. Die Definition einer Variablen darf an einer beliebigen Stelle innerhalb des Programms erfolgen.

Typ	**Inhalt**	**Initialer Wert**	**Größe (bit)**	**Wertebereich**
boolean	true oder false	false	1	keine Angaben
char	Unicode-Zeichen	\\u0000	16	\\u0000 bis \\uFFFF
byte	Integer mit Vorzeichen	0	8	-128 bis 127
short	Integer mit Vorzeichen	0	16	-32768 bis 32767
int	Integer mit Vorzeichen	0	32	-2147483648 bis 2147483647
long	Integer mit Vorzeichen	0	64	-9223372036854775808 bis 9223372036854775807

Typ	Inhalt	Initialer Wert	Größe (bit)	Wertebereich
float	IEEE 754 Fließkomma	0.0	32	1.40239846E-45 bis 3.40282347E+38
double	IEEE 754 Fließkomma	0.0	64	4.94065645841246544E-324 bis 1.79769313486231570E+308

Tabelle 1 Standarddatentypen

Java bietet alternativ zu jedem Standarddatentyp auch eine Java-Klasse an. Damit ist eine reine obejktorintierte Programmierung möglich, allerdings auf Kosten der Performance. Daher sollten wenn möglich die Standardtypen benutzt werden und nur für bestimmte Funktionen (z.B. spezielle Datentypkonvertierungen) auf die Klassen zurückgegriffen werden.

3.2 Operatoren

Operatoren werden auf Konstante bzw. Variable angewendet. Tabelle 2 enthält alle Java-Operatoren. Als Ergebnis entsteht stets ein neuer Wert, welcher z.B. einer Variablen zugewiesen werden kann.

Der Datentyp des Ergebnisses hängt zum einen vom Datentyp der beteiligten Operanden und vom Operator ab.

Beispiel

```
int a = 1, b = 2;

int c = a + b;
float d = x + y;     //Ergebnis wird typkonvertiert

boolean e = a > b;
```

Operator	Bedeutung

Operator	**Bedeutung**
X + Y	addieren oder verketten
X – Y	substrahieren
X * Y	multiplizieren
X / Y	dividieren
X % Y	modulus
X ^ Y	Exklusives oder
X & Y	bitweises und
X | Y	Bitweises oder
X && Y	logisches und
X || Y	logisches oder
X << Y	Linksverschiebung
X >> Y	Rechtsverschiebung
X >>> Y	vorzeichenlose Rechtsverschiebung
X = Y	Zuweisung
X *op*= Y	Operator ausführen und Ergebnis zuweisen

Operator	**Bedeutung**
X < Y	kleiner als
X > Y	größer als
X <= Y	kleiner gleich
X >= Y	größer gleich
X == Y	gleich
X!= Y	ungleich
X.Y	. Operator z.B.: java.lang.String
X, Y	Verkettung
X instanceof Y	Ist X Subklasse von Y?
X ? Y: Z	if then else
X++	inkrementieren postfix
X --	dekrementieren postfix
++ X	inkrementieren präfix
-- X	dekrementieren präfix
! X	nicht

Operator	Bedeutung
(X)	Vorrang
(Typ)X	Erzwungenene Typkonvertierung (cast)
new X	neue Instanz

Tabelle 2 Operatoren

3.2.1 Vorrang der Operatoren

- Reihenfolge: * / % vor + -
- bei gleicher Rangfolge erfolgt Abarbeitung von links nach rechts
- Ausdrücke in Klammern werden immer zuerst bewertet

Beispiel

```
//Der Ausdruck
a * b + a / b
//entspricht
(a * b) + (a / b)
```

Besonders bei ganzzahligen Operationen kann die Reihenfolge der Bearbeitung entscheidend sein.

Beispiel

```
int a=1, b=3, c=6, x;
x = a / b * c;          //Ergebnis: x = 0
x = a * c / b;          //Ergebnis: x = 2
```

3.2.2 Zuweisungsoperator

Eine spezielle Operatorform ist der Zuweiungsoperator der einzig dafür verwendet werden kann, einen Operatorausdruck kompakter zu formulieren.

Syntax: *operand1 operator= operand2*

Dies ist äquivalent zu:

operand1 = operand1 operator operand2

Beispiele

```
x = x + 5;
//kann ersetzt werden durch
x += 5;

//Der Ausdruck
x = x / (a + b)
//wird vereinfacht durch
x /= a + b;
```

3.3 Schleifen

Schleifen ermöglichen ein mehrfaches Durchlaufen gleicher Anweisungen. Die Kontrolle der Anzahl der Durchläufe erfolgt über einen logischen Ausdruck. Tabelle 3 zeigt die verschiedenen Schleifentypen von Java.

Typ	Syntax

for	`for(Anweis1;log. Ausdruck;Anweis2)`
	`{`
	`  Anweisungen`
	`}`

while	`while (log. Ausdruck)`
	`{`
	`  Anweisungen`
	`}`

do-while	`do`
	`{`
	`  Anweisungen`
	`}`
	`while (log. Ausdruck);`

Tabelle 3 Schleifen

Prinzipiell wäre ein einziger Schleifentyp ausreichend, da jede Art von Schleife mit allen 3 Typen nachgebildet werden kann. Für einige spezielle Anwendungen ergeben sich jedoch Vorteile für den einen oder anderen Schleifentyp.

3.3.1 for-Schleife

Die for-Schleife ist dann von Vorteil, wenn die Anzahl der Schleifendurchläufe bekannt ist und diese auch von den Anweisungen innerhalb der Schleife nicht beeinflußt wird.

Beispiel: Summe der Zahlen von 1 bis 100

```
int summe = 0;
for (int zahl=1; zahl<=100; zahl++)
{
   summe += zahl;
}
```

for-Anweisung nicht durch ; abschließen

```
int summe = 0;
for (int zahl=1; zahl<=100; zahl++);
//Schleife ist hier zu Ende!!
{
   summe += zahl;       //summe erhält hier den Wert von 101
}
```

3.3.2 while-Schleife

die while-Schliefe bringt Vorteile, wenn die Anzahl der Schöiefndurchläufe zunächst unbekannt ist und durch die Anweisungen innerhalb der Schleife bestimmt wird.

Beispiel: Summe der natürlichen Zahlen bis zu einem Maximum der Summe von 100

```
int zahl = 1, summe = 0;
while (summe <= 100)
{
   summe += zahl;
   zahl++;
}

//oder alternativ
while (summe <= 100)
   summe += zahl++;
```

while-Anweisung nie durch ; abschließen

```
int zahl = 1, summe = 0;
while (summe <= 100);     //Endlose Schleife, da summe stets 0
   summe += zahl++;
```

3.3.3 do-while-Schleife

Diese Schleife ist anlalog der while-Schleife, jedoch erfolgt die Prüfung auf Schleifenende erst **nach** dem Durchlauf der Anweisungen. Daher ist diese Form zu bevorzugen, wenn die Schleifenanweisungen mindestens einmal ausgeführt werden müssen (z.B. bei iterativen Verfahren).

Beispiel: Bestimmen der Zahl, welche bei der Summation dazu führt daß die Summe von 100 überschritten wird

```
int zahl=0, summe=0;
do
{
   summe += ++zahl;
} while (summe <= 100);        //Hier muß das ; stehen
```

3.3.4 Spezielle Schleifenanweisungen

Es existieren 2 spezielle Anweisungen zur Kontrolle eines Schleifendurchlaufs, die **break**- und die **continue**-Anweisung. Diese sollten jeodch sehr sparsam eingesetzt werden, da sie das Verständnis für den Programmablauf erschweren. Sie kommen vorwiegend dort zum Einsatz, wo infolge unvorhersehbarer Ereignisse (z.B. Fehlerzustände) eine weitere Bearbeitung der Anweisungen keinen Sinn ergeben.

Eine **break**-Anweisung beendet sofort die gesamte Schleife. Es wird mit der ersten Anweisung nach der Schleife fortgefahren.

Die **continue**-Anweisung übergeht die noch ausstehenden Anweisungen innerhalb der Schleife und beginnt sofort einen neuen Schleifendurchlauf.

3.4 Kontrollstrukturen (Ablaufkontrolle)

Kontrollstrukturen werden dazu verwendet, nur bestimmte Teile eines Programms auszuführen. Die Ausführung erfolgt dabei stets in Abhängigkeit vom Ergebnis eines logischen Ausdrucks. Tabelle 4 zeigt die verfügbaren Strukturen.

Art	**Schlüsselwort**	**Beispiel**

Entscheidung	if-else	`if (logischer Ausdruck)`
		`  {Anweisungen}`
		`else if (logischer Ausdruck)`
		`  {Anweisungen}`
		`else`
		`  {Anweisungen}`

Entscheidung	switch-case	`switch (Konstante)`
		`{`
		`  case Wert:`
		`  Anweisungen`
		`  break;`
		`  case Wert:`
		`  Anweisungen`
		`  break;`
		`  default:`
		`  Anweisungen`
		`}`

Art	Schlüsselwort	Beispiel
Ausnahme	try-catch	`Siehe 3.9`

Tabelle 4 Kontrollstrukturen

3.4.1 Bedingte Anweisung (if)

Anweisungen werden nur dann ausgeführt, wenn sich die zugehörige Bedingung (als logischer Ausdruck) zu wahr ergibt. Dabei können beliebig viele Bedingungen angegeben werden, welche nacheinander solange geprüft werden bis sich eine dieser Bedingungen als wahr erweist.

Syntax

```
if (bedingung1)
{
  Anweisungen 1
}
else if (bedingung2)
{
  Anweisungen 2
}
else if (bedingung3)
{
  Anweisungen 3
}
....
else
{
  Anweisungen N
}
```

Nur der erste if-Block ist zwingend erforderlich. Alle else if-Blöcke sowie der abschließende else-Block sind wahlweise. Es darf höchstens einen else-Block geben.

Beispiel

```
int i, a , b;
...
if (i > 0)
  a = b;
else if (i == -1)
  a = -b;
else
  a = 0;
```

In diesem Beispiel ist sichergestellt, daß der Variablen a genau einmal ein Wert zugewiesen wird, jeweils in Abhängigkeit des Wertes von i.

3.5 Schlüsselwörter

Folgende Schlüsselwörter sind in Java definiert und dürfen daher nicht zur Definition von Klassen, Funktionen bzw. Variablen benutzt werden.

abstract	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
var	void	volatile	while	

3.6 Escape-Sequenzen (in Zeichenketten)

Darstellung	**Bedeutung**

Darstellung	Bedeutung
\n	new line
\r	cariage return
\b	backspace
\t	tabulator
\f	form feed
\'	Hochkomma
\"	Anführungszeichen
\\	backslash
\uXXXX	Unicode-Zeichen

3.7 Unterschiede zu C,C++

Java konnte ohne Rücksicht auf Kompatibilität zu bereits vorhandenen Sprachen neu entwickelt werden. Dadurch wurden einerseits besonders fehlerträchtige „Features“ aus C und C++ nicht implementiert, andererseits aber auch neue Features integriert.

3.7.1 Fehlende Features

- **Zeiger**

  In Java werden prinzipiell Referenzen benutzt.

- **globale Funktionen**
  Es gibt in Java keine globalen Funktionen mehr. Funktionen sind stets Memberfunktionen, d.h. sie gehören immer zu einer Klasse. Als Ersatz für globale Funktionen können statische Memberfunktionen verwendet werden, die ohne ein existierendes oder bekanntes Objekt aufgerufen werden können

- **typedef , Strukturen, Unions**
  Muß in Java über Klassendefinition erfolgen

- **Präprozessor (#define,#ifdef ...)**
  Ist aufgrund der Plattformunabhängigkeit von Java nicht mehr erforderlich. Damit fällt auch die #include-Anweisung weg. Java besitzt einen eigenen Mechanismus, um Beziehungen zwischen Klassen zu berücksichtigen.

- **Mehrfachvererbung**
  Leider. Ist wohl nicht einfach (sauber) zu implementieren.

- **Überladen von Operatoren**
  Dies muß vollständig über Methoden implementiert werden

3.7.2 Neue Features

- **Threads**
  Sehr einfache Möglichkeit der Programmierung paralleler Prozesse

- **Speicherverwaltung (Garbage collection)**
  Nicht mehr benötigte Objekte werden automatisch entfernt

- **Sicherheit**
  Anwendungen können nicht mehr automatisch alle Systemfunktionen nutzen. Die Sicherheitsüberprüfung erfolgt direkt vor der Ausführung des Codes, nicht beim Kompilieren. Dadurch erhöhter Schutz auch vor Codemanipulationen.

- **Portabilität**
  Java ist vollständig plattformunabhängig, und das nicht nur im Source-Code

- **Unicode-Zeichensatz**
  Auch Umlaute können nun zur Programmierung genutzt werden. (Vorsicht: Nicht alle Browser untesterstützen Java-Applets mit Umlauten in Klassen und/oder Funktionen; z.B. erzeugt Netscape Communicator eine „Security-Violation“)

- **Umfangreiche Packages**
  Zu vielen Bereichen (Grafik, Netzwerk, Verschlüsselung etc.) existieren sehr umfangreiche Klassenbibliotheken (vgl. 3.8)

- **Netzwerkprogrammierung**
  Durch Package-Unterstützung sehr einfach möglich

3.8 Packages

Wie in anderen Programmiersprachen so existiert auch in Java ein Namensraum (Namespace) für Variable, Methoden, Objekte und Klassen. Insbesondere ist darauf zu achten, daß alle Klassennamen innerhalb einer Anwendung eindeutig sind (globaler Namensraum). Das führt meistens dann zu Problemen, wenn unterschiedliche Klassenhierarchien innerhalb eines Projekts vereinigt werden sollen.

Beispiel: 2 unabhängige Entwickler stellen je eine Klassenbibliothek zur Verfügung, innerhalb derer Klassen mit dem Namen „Object“ und „Vector“ vorkommen. Bei der Benutzung beider Klassenhierarchien kann der Compiler nicht entscheiden, welche der beiden Implementierungen nun im Einzelfall eingesetzt werden soll.

Daher gibt es in Java die Packages (ähnlich Modula oder Ada). Ein Package dient dabei lediglich der Gruppierung von Klassen unter einem einheitlichen Namensraum. Der Namensraum wird dabei einfach über ein Dateiverzeichnis abgebildet.

Beispiel: Die Klasse „Vector“ innerhalb des Package „java.util“ liegt als Datei „Vector.class“ innerhalb des Verzeichnisses „java/util“.

Durch Einbeziehung des Packagenamens als Bestandteil des Klassennamens können somit sehr einfach Namenskonflikte beseitigt werden. Es ist dazu lediglich erforderlich, daß die Klassen innerhalb eines Package definiert werden. Dies erfolgt durch die Angabe von:

package *name;*

jeweils in der ersten Zeile einer Java-Datei. Dabei muß jedoch sichergestellt werden (durch entsprechende Compiler-Optionen), daß die übersetzten Dateien (*.class) in dem durch ***name*** angegebenen Verzeichnis auch abgelegt werden. Wird kein **package** angegeben, so befinden sich allen in dieser Datei definierten Klassen im default-Package (besitzt keinen Namen).

Die Benutzung einer Klasse innerhalb eines Package erfolgt durch

import klassenname;

Sollen alle Klassen eines Package benutzt werden so ist das gesamte Package zu importieren:

import packagename.*;

Es erfolgt dabei aber kein rekursives Importieren, d.h. nur die nächste Stufe wird dabei importiert. Es hat sich als Konvention herausgebildet, als Packagenamen den Domainnamen der Firma in umgekehrter Reihenfolge zu setzen. So könnte die Entwicklung einer Klasse „Grafik“ im Package „de.unibw-muenchen.bauv.Graphics“ erfolgen. Durch die Verwendung des Domainnamens werden Namenskonflikte auf den Domainbereich reduziert.

Das Auffinden von verschiedenen Packages zur Laufzeit einer Anwendung wird über die Systemvariable **CLASSPATH** gesteuert. CLASSPATH zeigt auf den Beginn der Verzeichnisse, ab denen die Klassen zu den verschiedenen Packages abgelegt sind.

Beispiel: Befindet sich die Klasse „.Graphics.Grafik“ im Verzeichnis „D:/Software/Java/de/unibw-muenchen/bauv/Graphics“ und lautet die import-Anweisung „import Graphics.*;“ so muß CLASSPATH das Verzeichnis „D:/Software/Java/de/unibw-muenchen/bauv“ beinhalten.

Die CLASSPATH-Variable besitzt keine Bedeutung bei der Anwendung von Applets. Da aus Sicherheitsgründen Applets keinen direkten Zugang zu Verzeichnissen besitzen, werden alle benötigten Klassen vom WWW-Server geladen. Die Packages müssen sich dazu in einem Verzeichnis relativ zum HTML-Dokument befinden, welches das Applet-Tag enthält.

Beispiel: Benutzt ein Applet die Klasse „de.unibw-muenchen.bauv.Graphics.Grafik“ (über die import-Anweisung "import de.unibw-muenchen.bauv.Graphics.*") so muß sich relativ zum HTML-Dokument das Verzeichnis „de/unibw-muenchen/bauv/Graphics“ befinden.

3.8.1 Vorhandene Packages

Java beinhaltet bereits viele Packages zur Unterstützung bei der Programmierung unterschiedlichster Anwendungen. Die wichtigsten Packages bzw. Klassen innerhalb dieser Packages sind:

Package	Klasse	Anwendung
java.lang		Standardpackage, wird immer importiert
	Object	Basisklasse aller Java-Klassen
java.awt		Abstract Window Toolkit zur Erzeugung von Benutzeroberflächen.
	Graphics	einfache 2D-Grafikfunktionen
java.util		Allgemeine verwendbare Klassen
	String	Zeichenketten
	Vector	zur Aufnahme beliebiger Objekte, besitzt varibale Länge und ist dynamisch veränderbar
	Hashtable	wie Vektor, jedoch besitzen die Objekte einen eindeutigen Schlüssel (der Klasse Object) über den sie sehr schnell wieder auffindbar sind.
	Random	Zufallszahlen

Package	Klasse	Anwendung
	Date	Zeit- und Datumsfunktionalität
java.sql		SQL-Unterstützung zur Anbindung relationaler Datenbanken
java.rmi		„Remote Method Invocation“ zur Realisierung verteilter Anwendungen im Netzwerk
com.sun.j3d		sehr komfortable 3D-Bibliothek zur Erstellung virtueller Welten (derzeit Beta)

3.8.2 Beispiel zur Nutzung von Klassen verschiedener Packages

```
//Nutzung verschiedener Packages

import java.applet.*;
import java.awt.*;
import java.util.*;

public class TestPackages extends Applet
{
  public TestPackages ()
  {
    //Definition von Linien
    for (int i=1; i<=zahl()/5; i++)
    {
      //neues Linienobjekt
      Linie l = new Linie ();

      //Linie dem Vektor hinzufuegen
      objekte.addElement(l);
    }

    //Definition von Kreisen
    for (int i=1; i<zahl()/10; i++)
    {
      Kreis k = new Kreis();
      objekte.addElement(k);
    }
  }

  public void paint(Graphics g)
  {
    //Ausgabe der aktuellen Zeit
    Date datum = new Date();

    String ausgabe = new String("Aktuelle Zeit: " + datum);
    g.drawString (ausgabe, 0, 50);

    //Zaehlen der Objekte
    int kreise = 0, linien = 0;

    for (int i=0; i<objekte.size(); i++)
    {
      Object o = objekte.elementAt(i);

      if (o instanceof Linie)
      {
        linien++;
        g.setColor(Color.red);
        Linie l = (Linie) o;
        g.drawLine(l.x1, l.y1, l.x2, l.y2);
      }
      else if (o instanceof Kreis)
      {
        kreise++;
        g.setColor(Color.blue);
        Kreis k = (Kreis) o;
        g.drawArc(k.x-k.r, k.y-k.r, 2*k.r, 2*k.r, 0, 360);
        g.drawString(""+k,k.x,k.y);
      }
    }

    g.setColor(Color.black);
    g.drawString("Linien: " + linien + ", Kreise: " + kreise,
0, 80);
  }

  public static int zahl()
  {
    //Liefert ganze Zahl in [0,300]
```

```
     int x = zufall.nextInt();
     //x positiv und < 300
     if (x < 0)
       x = -x;
     while (x > 300)
       x -= 300;

     return x;
   }

   //Vector zur Aufnahme aller Objekte
   private Vector objekte = new Vector();
   public static Random zufall = new Random();
}

class Linie
{
   public Linie()
   {
     //Werte aus Zufallszahlen
     x1 = TestPackages.zahl();
     x2 = TestPackages.zahl();
     y1 = TestPackages.zahl();
     y2 = TestPackages.zahl();
   }

   public int x1, x2, y1, y2;
}

class Kreis
{
   public Kreis()
   {
     //Werte aus Zufallszahlen
     x = TestPackages.zahl();
     y = TestPackages.zahl();;
     r = TestPackages.zahl() / 10;
   }

   public int x, y, r;
}
```

Die Besonderheiten bei diesem Beispiel liegen in der Verwendung von:

- static

  Das Objekt "zufall" sowie die Funktion "zahl" sind als static definiert, d.h. sie gehören zur Klasse und nicht zum jeweiligen Objekt. Damit können sie auch ohne ein existierendes Objekt über den Klassennamen aufgerufen werden.

- instanceof

  Der Vektor "objekte" speichert Objekte des Typs "Object", d.h. alle Java-Objekte (die als Basisklasse per Definition "Object" besitzen) können hier abgelegt werden. Über das Kennwort "instanceof" kann festgestellt werden, um welchen Objekttyp es sich handelt.

- Zeichenketten (String)

  Die Verarbeitung von Zeichenketten ist in Java sehr einfach. Zeichenketten können einfach mit "+" aneinandergefügt werden. Jedes Objekt (durch die Basisklasse "Object") besitzt eine textuelle Darstellung.

3.9 Ausnahmebehandlung (Exceptions)

Innerhalb eines Programms treten sehr häufig Zustände auf, die besonders beachtet werden müssen, so daß sie keine fehlerhaften Anweisungen verursachen. Die häufigsten Probleme sind:

- Indizierung eines Vektor-Elementes außerhalb der gültigen Grenzen

```
int daten[2];
...
int x = daten[2];  //Fehler!
```

- Referenzierung eines nicht existierenden Objektes

```
Vector objekte;  //besitzt keine Referenz
...
Object o = objekte.elementAt(i);
```

- Versuchte Umwandlung eines Objektes in einen ungültigen Datentyp

```
Vector objekte = new Vektor();
objekte.addElement(new Kreis());
Object o = objekte.elementAt(0);
Linie l = (Linie) o; //Fehler, 1. Element ist vom Typ Kreis
```

- Fehler durch Systemfunktionen

```
File datei = new File("/tmp/foo");
myFile.delete();   //Fehler bei Applet (vgl. 3.10)
```

Einige dieser Probleme lassen sich durch geeignete Form der Programmierung (if-Bedingungen etc.) lösen. Allerdings bietet Java dafür einen besonderen Mechanismus an, die Exceptions.

Syntax:

```
try
{
   //kritischer Code
}
catch (Exception1 e1)
{
   //Behandlung e1
}
catch (Exception2 e2)
{
```

```
    //Behandlung e2
}
...
```

Sobald eine Ausnahme (Exception) innerhalb des try-Blocks auftritt, wird dieser Block verlassen und der entsprechende catch-Block (je nach Typ der Ausnahme) aktiviert. Ist keine spezielle Behandlung erforderlich, so genügt:

```
try
{
    //kritischer Code
}
catch (Exception e)
{}
...
```

Der Vorteil des Arbeitens mit Exceptions (anstatt der vielleicht möglichen Überprüfung durch if-Anweisungen) liegt in der klaren Strukturierung. Die Anweisungen werden ohne Rücksicht auf Fehlermöglichkeiten programmiert, die Fehlerüberprüfung erfolgt an einer zentralen Stelle.

Es können jederzeit auch eigene Exceptions definiert und ausgelöst werden. Dazu muß die auslösende Funktion durch

typ name (parameter) throws Exception

definiert werden. Innerhalb dieser Funktion kann dann die Ausnahme durch

throw ExceptionObjekt;

ausgelöst werden. "ExceptionObjekt" ist ein von der Klasse "Exception" bzw. von einer davon abgeleiteten Klasse instanziiertes Objekt.

```
//Test Exceptions
import java.applet.*;
import java.util.*;
import java.awt.*;

public class TestExceptions extends Applet
{
   public void paint(Graphics g)
   {
      try
      {
         test1();
         g.drawString("Arbeiten erlaubt",10,10);
         float x = test2(5, 3);
         g.drawString("Ergebnis 1 = " + x, 10,20);
         x = test2(5, 0);
         g.drawString("Ergebnis 2 = " + x, 10,30);
      }
      catch (Exception e)
      {
         g.setColor(Color.red);
         g.drawString("Fehler: " + e, 10, 100);
      }
   }

   private void test1() throws Exception
   {
      Date datum = new Date();

      if (datum.getHours() < 8 || datum.getHours() > 17)
      {
         Exception feierabend = new Exception("Keine Arbeit
außerhalb der Bürozeiten!");
         throw feierabend;
      }

      //Erlaubnis zur Arbeit...
   }

   private int test2(int a, int b) throws Exception
   {
      int division;

      try
      {
         division = a / b;
      }
      catch (ArithmeticException e)
      {
         throw e;
      }

      return division;
   }
}
```

Das Beispiel zeigt, daß die Fehlerbehandlung sehr einfach ist. Insbesondere ist es bei Funktionen sinnvoll, denen man "von außen" ein erfolgreiches Arbeiten nicht ansieht (z.B. Funktion "test2").

Die Sicherheit eines Java-Programms ist auch dadurch gewärleistet, daß evtl. auftretende Ausnahmen auch stets behandelt werden müssen, d.h. der Aufruf einer Funktion, welche eine Ausnahme auslösen kann, muß durch try-catch erfolgen.

3.10 Java-Sicherheitskonzept

Insbesondere durch die Möglichkeit, Java-Klassen über das Netzwerk zu laden und auszuführen ist ein Sicherheitskonzept erforderlich und in Java auch implementiert. Dabei wird nach dem Laden des Bytecodes vor der eigentlichen Ausführung eine Sicherheitsüberprüfung durchgeführt. Diese Überprüfung findert nicht nur unzulässige Anweisungen im Code sondern auch etwaige Codemanipulationen, woraufhin die Ausführung des Bytecodes verweigert wird. Nachfolgende Tabelle zeigt die zulässigen Operation in Abhängigkeit der verwendeten Umgebung.

	Browser[1] Bytecode über Netz	**Browser** Bytecode über lokales Filesystem	**Appletviewer**[2] Bytecode über Netz	**Appletviewer** Bytecode über lokales Filesystem	**Java-Applikation**
Datei[3] auf Festplatte lesen	✘	✘	✘		
Datei auf Festplatte schreiben	✘	✘	✘		
Informationen über Dateien	✘	✘	✘		
Dateien löschen	✘	✘	✘	✘	

[1] Beispiel: Netscape Communicator

[2] Sun's "appletviewer"

[3] Beim Appletviewer kann der Zugriff auf Dateien explizit über ACL-Listen (Acces Control Lists) erlaubt werden.

	Browser[1] Bytecode über Netz	**Browser** Bytecode über lokales Filesystem	**Appletviewer**[2] Bytecode über Netz	**Appletviewer** Bytecode über lokales Filesystem	**Java-Applikation**
Benutzername erhalten	✘		✘		
Verbindung zu Port an Client	✘	✘	✘		
Verbindung zu einem dritten Rechner	✘	✘	✘		
exit-Funktion aufrufen	✘	✘	✘		

Wird eine nicht erlaubte Funktion dennoch versucht, so erfolgt eine Security-Exception.

```
//Erfragen des Benutzernamens
String benutzer;
try
{
   benutzer = System.getProperty("user.name");
}
catch (SecurityException e)
{
   benutzer = new String("Unbekannt");
}
```

4 Konzepte der objektorientierten Programmierung (OOP)

4.1 Begriffe der OOP

Begriff	**Erläuterung**	**Beispiel[e]**
Klasse	„Abstrakter“ Typ. Es erfolgt quasi eine Typisierung anhand charakteristischer Merkmale wie Eigenschaften und Fähigkeiten. Klassen sind wichtigster Bestandteil der objektorientierten Betrachtungsweise. Es erfolgt eine Aufteilung der „realen“ Welt in Klassen. Diese Aufteilung läßt sich beliebig fein durchführen. Im Idealfall sind die Klassen schon vorgegeben (z.B. durch eine Designstudie) und können dann einfach benutzt werden	Auto Farbe Tankinhalt Geschw. kann fahren muß tanken etc.
Objekt (Instanz)	Dieses sind real existierende Objekte, d.h. Objekte mit denen im Verlauf der Programmierung gearbeitet werden kann. Objekte beziehen sich stets auf Klassen (eine Klasse gibt also den Typ der Objekte vor).	polo mercedes etc.
(Member-) Funktionen (Nachrichten, Messages)	Dienen dazu, bestimmte Fähigkeiten der Klasse zu implementieren (z.B. ein Auto kann fahren) Memberfunktionen sind stets einem Objekt zugeordnet. Ein Objekt reagiert auf eine solche Nachricht mit einer bestimmten Aktion. Als Trennzeichen zwischen Objekt und Funktion dient der	mercedes.info() polo.fahren(100)

Begriff	**Erläuterung**	**Beispiel[e]**
	'.' . Funktionen können (innerhalb der '()') weitere Argumente (Parameter) besitzen, welche genauere Anweisungen beinhalten.	
(Member-) Variable	Dienen dazu, bestimmte Eigenschaften eines Objekts dieser Klasse zu verwalten (z.B. die Eigenschaften Farbe und Tankinhalt eines bestimmten Autos). Membervariablen sind wie die Funktionen stets einem Objekt zugeordnet. Zugriff auf diese Variablen erfolgt in der Regel über Memberfunktionen, jedoch kann u.U. auch direkt darauf zugegriffen werden.	polo.tankinhalt = 100;
Zugriffs-regelungen (Sicherheit)	Auf öffentliche Funktionen bzw. Variablen eines Objekts darf ohne Einschränkung zugegriffen werden. Auf private Funktionen bzw. Variable darf nur das Objekt selbst zugreifen. In der Regel existiert keine Unterscheidung zwischen lesendem und schreibendem Zugriff.	

Beisspiel. Klasse „Linie“ (siehe 2.1)

```
public class Linie
{

   //Konstruktor
   public Linie( int x1, int y1, int x2, int y2)
   {
      this.x1 = x1;
      this.x2 = x2;
      this.y1 = y1;
      this.y2 = y2;
   }

   //Methoden (Zeichnen der Linie)
   public void zeichnen(Graphics g)
   {
      g.drawLine (x1, y1, x2, y2);
   }


   //Variablen (Anfangs/Endpunkt einer Linie)
   private int x1 , y1 , x2 , y2;
}
```

Hier wird eine Klasse **Linie** deklariert. Diese besitzt die allgemeinen Eigenschaften x1, y1, x2, y2, welche die x,y-Koordinate des Anfangs- und Endpunktes der Linie darstellen. Da diese Variablen als **private** deklariert sind, kann darauf nie direkt zugegriffen werden.

Desweiteren besitzt eine Linie (genauer gesagt: jedes Objekt der Klasse Linie) die Fähigkeit, sich selbt grafisch darszustellen. Dies erfolgt über die Funktion **zeichnen**.

Damit können beliebige Linienobjekte erzeugt (instanziiert) werden, z.B:

```
Linie l1 = new Linie(0,0, 100,100);
Linie l2 = l1;
Linie l3;
```

Das Objekt l1 definiert ein eigenständiges (neues) Linienobjekt mit den Koordinaten (0,0) und (100,100). Das Objekt l2 ist kein neues Objekt, da es sich lediglich auf das Objekt l1 bezieht (Referenz), d.h. die Verwendung des Namens l2 ist ab hier identisch mit dem Namen l1. Das Objekt l3 ist eine nicht initialisierte Linie (es fehlt der Operator new). l3 besitzt somit den Wert **null** und jeglicher Zugriff auf dieses Objekt (z.B. der Versuch zu zeichnen) resultiert in einem Fehler.

4.2 (Member-)Funktionen

Hier wird auf Funktionsmechanismus unter Java generell eingegangen. Weitere Ausführungen zum Design von Klassen (und deren Memberfunktionen) siehe 4.5.

Eine Funktion läßt sich allgemein nach Abbildung 4-1 beschreiben. Bei der Aktivierung der Funktion (Funktionsaufruf) werden durch den Programmierer Parameter übergeben, welche innerhalb der Funktion dazu verwendet werden, die Eigenschaften des betreffenden Objekts zu benutzen oder zu manipulieren und ein Ergebnis der Funktion zurückzuliefern.

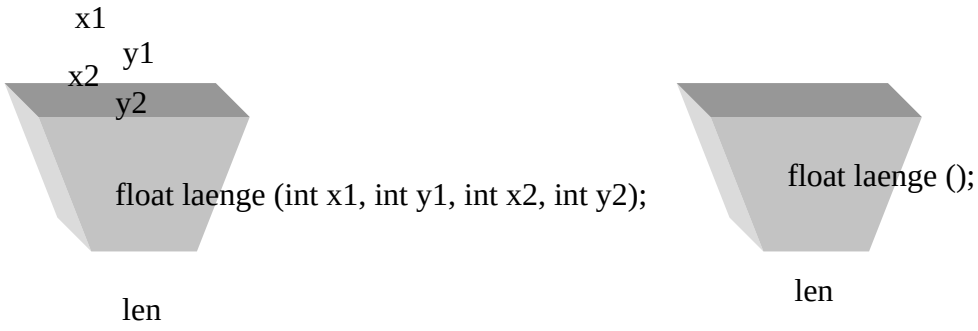


Abbildung 4-1 Funktion

Hier wird die Funktion **laenge** aufgerufen, welche als Ergebnis die berechnete Differenz zwischen den eingegebenen Koordinaten ermittelt und zurückliefert.

Für die Benutzung einer Funktion ist die Kenntnis der Aufrufsyntax (Signatur der Funktion) ausreichend. Interne Details der Implementierung sind hier völlig unwichtig. Es kann verschiedene Versionen einer Funktion geben, welche sich in ihrer Signatur unterscheiden (z.B. in Art und/oder Anzahl der verwendeten Parameter).

> Funktionen in Java sind immer Bestandteil einer Klasse, d.h. sie können auch nur über ein real exstsierendes Objekt einer solchen Klasse aufgerufen werden.
> (Ausnahme: statische Funktionen, siehe 4.3.6)

Beispiel zum Funktionsaufruf:

```
Linie l1 = new Linie (0,0, 100,100);

float len1 = l1.laenge (0,0, 100, 100);
float len2 = l1.laenge();

if (len1 != len2)
{
   //Fehler in der Implementierung ☹
   ...
```

Das Objekt *l1* wird zum Aufruf der Funktion *laenge* benötigt bzw. die Memberfunktion laenge der Klasse *Linie* wird für das Objekt *l1* aufgerufen.

4.3 Klassenkonzept (Vererbung)

4.3.1 Basisklasse (Superclass)

Die Basisklasse stellt allgemeine Eigenschaften für eine Gruppe gleichartiger Klassen zur Verfügung. Alle diese Eigenschaften werden bei einer späteren Vererbung an abgeleitete Klassen automatisch weitergegeben und müssen somit nicht neu definiert werden. Dies macht natürlich nur dann Sinn, wenn diese Klassen auch wirklich Gemeinsamkeiten besitzen.

Beispiel: Die Klasse „Fahrzeug“ definiert die allgemeinen Eigenschaften „besitzt Antrieb“, „kann fahren“ etc.

Bei der Implementierung können dabei auch Eigenschaften vorgegeben werden, die abgeleitete Klassen unbedingt besitzen und damit auch implementieren müssen.

4.3.2 Abgeleitete Klasse (Subclass)

Diese bezieht sich immer auf eine Basisklasse und erbt somit alle Eigenschaften dieser Klasse. Weitere Eigenschaften werden hier definiert (Verfeinerung, Spezialisierung) und es können bereits vorhandene Eigenschaften aus der Baisklasse neu implementiert (überladen) werden.

Beispiel: Die Klasse „Auto“ erbt von der Klasse „Fahrzeug“ alle Eigenschaften und definiert die Eigenschaften „Räder“, „Lenkrad“, „Motor“ etc. neu.

Jede Klasse (auch eine bereits abgeleitete Klasse) kann wiederum als Basisklasse verwendet werden.

Beispiel: Die Klasse „Lastwagen“ ist von der Basisklasse „Fahrzeug“ abgeleitet und definiert die zusätzlichen Eigenschaften „Ladefläche“, „Nutzlast“ etc.

4.3.3 Abstrakte Klasse (abstract class)

Eine solche Klasse kann niemals instanziiert werden, d.h. es dürfen keine Objekte dieser Klasse erzeugt werden. Dies wird insbesondere dann benutzt, wenn zunächst nur die

Fähigkeiten weiterer Klassen festgelegt werden sollen und diese Fähigkeiten dann in weiteren Klassen zu implementieren sind. Da diese Klassen quasi ein Protokoll für weitere Klassen darstellen, werden sie oft auch als „Protokollklassen“ (protocol classes) bezeichnet. Die später zu implementierenden Fähigkeiten werden als abstrakte Funktionen deklariert, d.h. eine abstrakte Klasse deklariert mindestens eine abstrakte Funktion und/oder die Deklaration einer abstrakten Funktion kann nur innerhalb einer abstrakten Klasse erfolgen.

4.3.4 Mehrfachvererbung (multiple inheritance)

Eine immer wieder heiß diskutierte Möglichkeit der Vererbung von Eigenschaften aus mehr als einer Basisklasse. Bei sinnvoller Anwendung (Abbildung 4-2) kann damit die Komplexität eines Programms deutlich verringert werden. Allerdings führt die mißbräuchliche Verwendung der Mehrfachvererbung oder bestimmte Aufbauten von Klassenhierarchien zu Mehrdeutigkeiten (Abbildung 4-3) und wohl aus diesem Grund fehlt dieses Feature in vielen objektorientierten Programmiersprachen, so auch in Java.

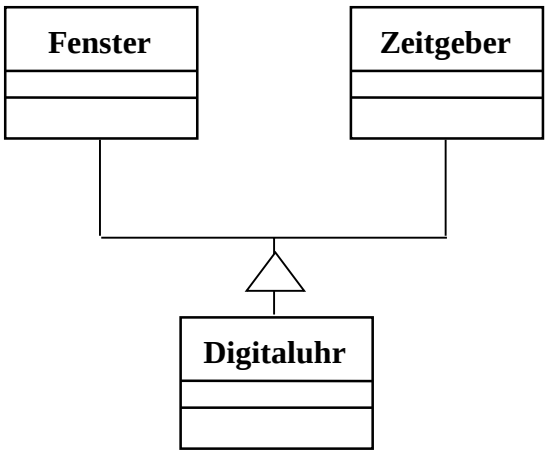


Abbildung 4-2 Mögliche Mehrfachvererbung

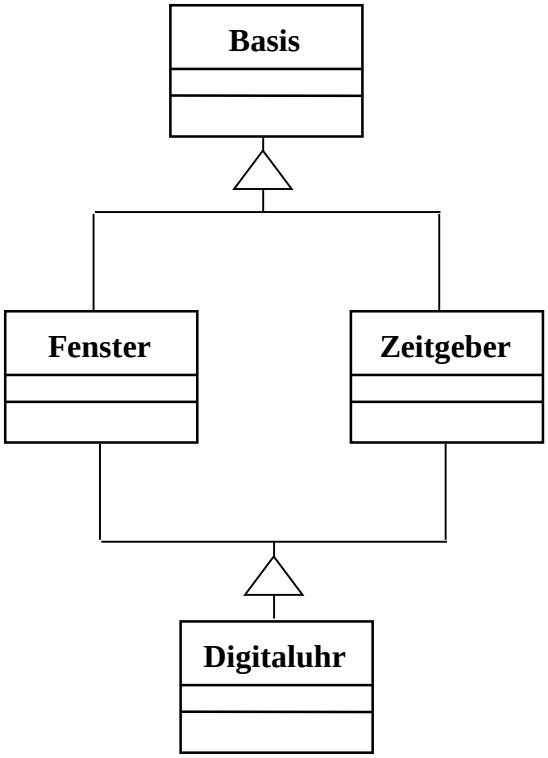


Abbildung 4-3 Mehrdeutigkeit bei Mehrfachvererbung

4.3.5 Schnittstellen (interfaces)

Diese definieren analog zu abstrakten Klassen Vorgaben für weitere Klassen, allerdings nicht über den Mechanismus der Vererbung. Schnittstellendeklarationen können von jeder beliebigen Klasse implementiert werden. Damit dürfen auch Klassen, die nicht in einer Vererbungsbeziehung zueinander stehen, die gleichen Schnittstellen implementieren. Eine Klasse kann auch mehrere Schnittstellen implementieren, weshalb dieses Vorgehen als (mangelhafter) Ersatz für das Fehlen von Mehrfachvererbung benutzt werden kann. Schnittstellendeklarationen können im Gegensatz zu abstrakten Klassen keine Implementierung vorgeben. Es wird also nur ein Funktionsprototyp vorgegeben. Die Implementierung muß stets in der Klasse erfolgen, welche die Schnittstelle implementiert.

4.3.6 Implementierung von Funktionen

Nachdem die Signatur einer Funktion (Art der Übergabeparameter sowie des Rückgabewerts, siehe auch 4.2) feststeht wird die Funktion innerhalb der zugehörigen Klasse implementiert. Diese Implementierung unetrscheidet sich nicht von der Implemntierung in herkömmlichen Programmiersprachen.

- Alle Übergabeparameter stehen als lokale Variable zur Verfügung

- Der Rückgabewert wird zweckmäßigerweise zunächst als lokale Variable definiert und am Ende der Funktion mittels **return** zurückgeliefert.

- Alle Parameter werden als lokale Kopie and die Funktion übergeben. Dies hat den gleichen Effekt, als würde man vor Aufruf der Funktion alle Parameter einer neuen Variablen zuweisen und dann diese neuen Variablen übergeben.

Beispiel: Implementierung der Funktion laenge (siehe 4.2)

```
public class Linie
{
   ...
   public float laenge (int x1, int y1, int x2, int y2)
   {
      float diffx = x2-x1, diffy = y2-y1;
      float len = Math.sqrt (diffx*diffx + diffy*diffy);

      return len;
   }

   public float laenge ()
   {
      return laenge(x1, y1, x2, y2); //Parameter der akt. Linie
   }
}
```

Die Implementierung einer Funktion ist also nichts anderes als die herkömmliche Programmierung eines Algorithmus unter der Verwendung bereits vordefinierter und initialisierter Variablen, der Funktionsparameter.

4.3.7 Statische Memberfunktionen

Während der Entwicklung einer komplexen Anwendung lassen sich viele Funktionen finden, die keinen direkten Bezug zu einem Objekt benötigen (analog den globalen Funktionen in anderen Programmiersprachen). Da jedoch in Java nur Memberfunktionen gestattet sind, kann dafür der Mechanismus der statischen Funktionen benutzt werden.

Statische Funktionen sind stets für alle Objekte der Klasse gemeinsam definiert, d.h. sie benötigen nur die Existenz einer Klasse und nicht die eines zugehörigen Objekts. Damit lassen sich diese Funktionen analog zu den nichtstatischen Memberfunktionen über die Klassenbezeichnung (anstatt der Objektbezeichnung) aufrufen.

4.4 Notation

Zu Zwecken der Dokumentation und der Übersicht ist es vorteilhaft, die Hierarchie der zu implementierenden Klassen zunächst in einer grafischen Notation festzuhalten. Dafür existieren verschiedene Methoden, alle mit Vor- und Nachteilen behaftet. Eine große Verbreitung besitzt die Object Modeling Technique (OMT) nach Rumbaugh. Daher soll im Folgenden diese Notation benutzt werden. Das Umsetzen auf andere Notationen ist i.d.R. einfach möglich, da sich diese im Wesentlichen nur in der verwendeten Symbolik unterscheiden. Die Notation ist (soweit in diesem Skript verwendet) in Kapitel 9.1 zusammengestellt.

4.5 Klassendesign

Anhand eines Beispiels soll ein Design mit der angesprochenen Notation durchgeführt werden. In der Literatur gibt es jede Menge Vorschläge, mit deren Hilfe ein objektorientiertes Design erfolgen kann. Hier wird ein mehr pragmatischer Ansatz gewählt, der sich aus verschiedenen Vorschlägen der Literatur aufbaut und sich (zumindest beim Autor) in zahlreichen Projekten bewährte. Die Zielsetzung dabei ist:

- Reduktion des Designs auf ein Minimum
- frühzeitige Implementierung von Prototypen
- Redesign, falls bei Implementierung der Prototypen Schwachstellen auftauchen

Die Vorgehensweise zeigt Abbildung 4-4

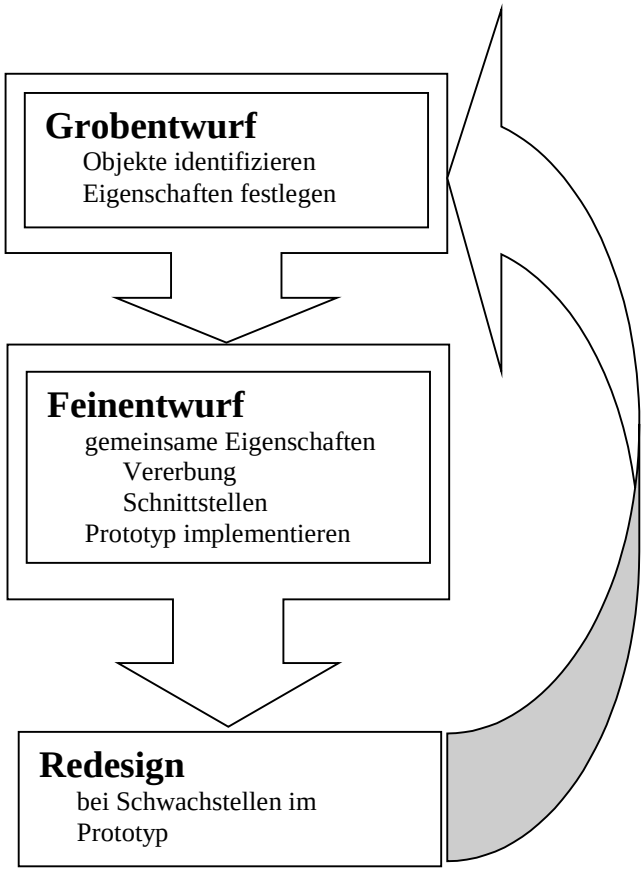


Abbildung 4-4 Entwurf Klassendesign

4.6 Realisierung in Java

Das Klassenkonzept nach 0 wird vollständig in Java unterstützt. Die Deklaration und Implementierung erfolgt stets gemeinsam innerhalb einer Datei, d.h. es gibt im Gegensatz zu C/C++ keine Headerdateien mehr. Der Dateiname ergibt sich aus dem Namen der öffentlich zugänglichen Klasse (Derzeit muß der Name der Java-Quellcodedatei mit dem Namen der darin definierten öffentlich zugänglichen Klasse übereinstimmen. In weiteren Java-Versionen wird diese Einschränkung in der Namensgebung entfallen!).

Die Syntax ergibt sich aus den folgenden Abschnitten. Dabei wird nachstehende Schreibweise benutzt:

- [ ] Angaben innerhalb der Klammern sind optional
- | Angaben sind entweder/oder

4.6.1 Klassendefinition

Syntax:

```
[public | protected] [abstract] class Klassenname
{
   Methoden
   Attribute
}
```

4.6.2 Methodendefinition

Syntax:

```
[public | protected | private] [abstract | static] typ name(parameter)
{
   Anweisungen;
}
```

Die Kennwörter besitzen folgende Bedeutung:

- **public**
  Die Klasse (Methode, Attribut) ist öffentlich, d.h. sie kann von beliebiger Stelle aus benutzt werden.

- **protected** (Default)
  Die Methode (Attribut) ist geschützt. Sie kann nur von abgeleiteten Klassen oder von Klassen innerhalb des gleichen Java-Package benutzt werden.

- **private**
  Die Methode (Attribut) ist privat. Sie kann nur von der Klasse selbst benutzt werden.

- **static**
  Die Methode (Attribut) ist statisch, d.h. es gehört ausschließlich zur Klasse, nicht zum instanziierten Objekt. Damit kann diese Methode (Attribut) auch ohne ein zugehöriges Objekt genutzt werden. Dies ist z.B. eine Möglichkeit zur Definition „globaler“ Methoden und Attribute.

- **abstract**
  Es handelt sich um eine abstrakte Methode einer abstrakten Klasse. Diese Methode muß in einer abgeleiteten Klasse zwingend implementiert werden, es sei denn, diese abgeleitete Klasse wird ebenso als abstrakte Klasse definiert. Abstrakte Methoden können nur in abstrakten Klassen deklariert werden.

4.7 Beispiel

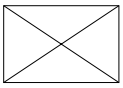
Es soll ein Java-Applet für ein einfaches Grafiksystem entwickelt werden. Die Entwicklung soll schrittweise erfolgen. In einem ersten Ansatz soll nur eine Basisfunktionalität bereitgestellt werden. In weiteren Kapiteln wird dieses System sukzessive erweitert um auch zusätzliche Java-Eigenschaften ausnutzen zu können.

4.7.1 Design

Analog zu Abbildung 4-4 erfolgt das Design in 3 Schritten.

4.7.1.1 Grobentwurf

Die Identifizierung von Objekten ist allein von der geforderten Funktionalität abhängig. in einem ersten Schritt sollen nur die Grafikelemente

- Linie 
- Rechteck 
- Kreis 
- Kasten 

implementiert werden. Die Fähigkeit der Objekte beschränkt sich zunächst nur auf die reine Darstellung. Damit ergeben sich die Objekte nach Abbildung 4-5

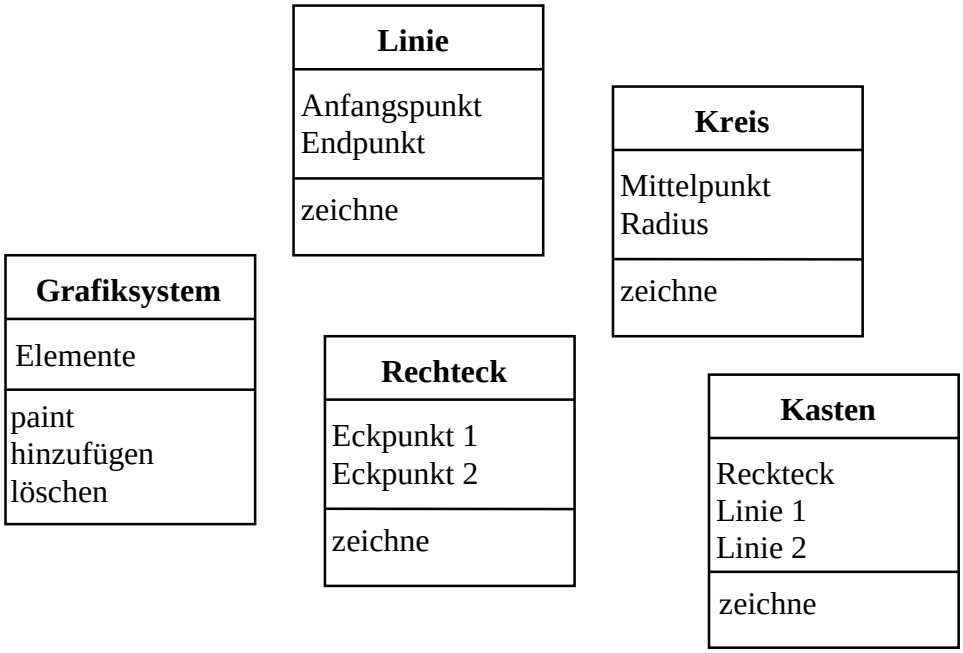


Abbildung 4-5 Identifizierung von Objekten

4.7.1.2 Feinentwurf

Bei der Analyse der Objekte nach Abbildung 4-5 ergeben sich folgende gemeinsame Merkmale.

- Alle grafischen Objekte implementieren eine Funktion „zeichne“ zur eigenen grafischen Darstellung. Diese Funktion ist zwingend erforderlich.
- Alle grafischen Objekte benutzen ein weiteres Objekt „Punkt“ zur Definition von Koordinaten.
- Das Objekt „Kasten“ benutzt die Objekte „Rechteck“ und „Linie“

Damit ergibt sich das Klassendesign nach Abbildung 4-6

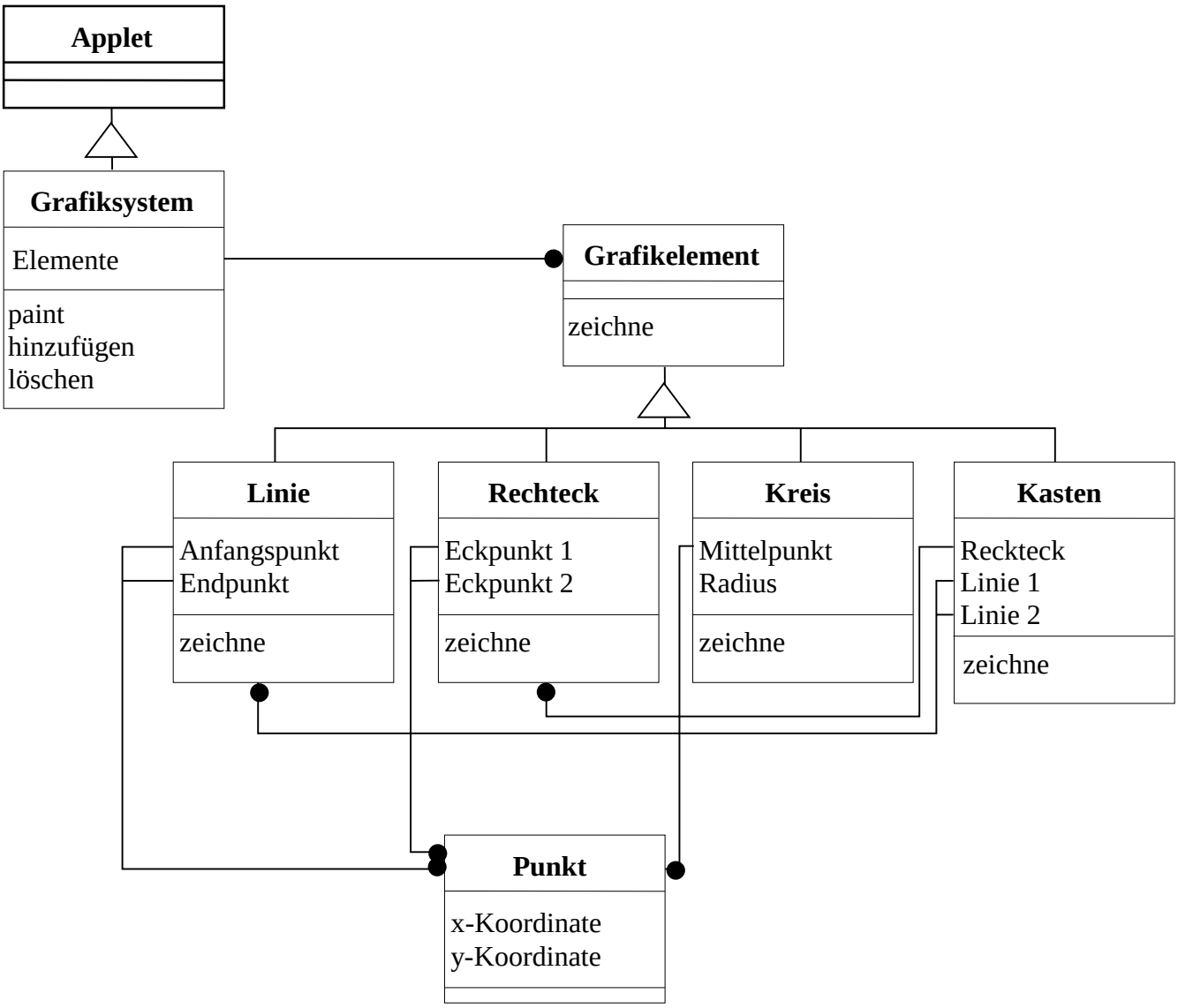


Abbildung 4-6 Klassendesign

4.7.1.3 Implementierung

Ein Klassendesign analog zu Abbildung 4-6 sollte stets unabhängig von einer Programmiersprache sein. Damit bleiben für die Implementierung noch einige Details offen. In unserem Beispiel gibt es für die Implementierung der Klasse „Grafikelement“ 2 Möglichkeiten:

- Implementierung einer Vererbung
  Um abgeleitete Klassen zu einer Implementierung der Methode „zeichne“ zu zwingen, sollte die Klasse sowie die Methode als abstract deklariert werden.

- Implementierung einer Schnittstelle

  Die Schnittstellenimplementierung ist deshalb möglich, da „Grafikelement“ keine Attribute besitzt und auch sonst keinerlei Methoden vordefiniert. Daher wird diese Klasse als Schnittstelle realisiert.

```
//"Grafikelement.java"
//Schnittstelle Grafikelement
import java.awt.*; //für Graphics

public interface Grafikelement
{
   public void zeichne (Graphics g);
}
```

Die grafischen Elemente werden alle innerhalb einer Datei implementiert. Dadurch dürfen die Klassen nicht als public definiert werden (zumindest nicht in der verwendeten Java-Version), da die Klassennamen vom Dateinamen abweichen. Da aber keine Packages definiert sind, sind diese Klassen allgemein zugänglich.

```
//"Grafik.java"
//Grafische Elemente

import java.awt.*;

//Punkt
class Punkt
{
   //Konstruktoren
   public Punkt(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   public Punkt()
   {}

   //Variable
   public int x, y;
}

//Linie
class Linie implements Grafikelement
{
   //Konstruktor
   public Linie (Punkt anfang, Punkt ende)
   {
      this.anfang = anfang;
      this.ende = ende;
   }

   //Implementierung der Schnittstelle Grafikelement
   public void zeichne (Graphics g)
   {
      g.drawLine (anfang.x, anfang.y, ende.x,ende.y);
   }

   //Variable
   Punkt anfang, ende;
}

//Rechteck
class Rechteck implements Grafikelement
{
   //Konstruktor
   public Rechteck (Punkt ecke1, Punkt ecke2)
   {
      e1 = ecke1;
      e2 = ecke2;
   }

   public void zeichne (Graphics g)
   {
      g.drawRect(e1.x,e1.y, e2.x-e1.x, e2.y-e1.y);
   }

   //Eckpunkte des Recktecks
   Punkt e1, e2;
}

//Kreis
class Kreis implements Grafikelement
{
   public Kreis (Punkt m, int r)
   {
      mittelpunkt = m;
      radius = r;
   }

   public void zeichne (Graphics g)
```

```
   {
     g.drawArc(mittelpunkt.x-radius, mittelpunkt.y-radius,
2*radius, 2*radius, 0, 360);
   }

   //Mittelpunkt und Radius
   Punkt mittelpunkt;
   int radius;
}

//Kasten
class Kasten implements Grafikelement
{
   public Kasten(Rechteck r)
   {
     rechteck = r;
     linie1 = new Linie(rechteck.e1, rechteck.e2);
     linie2 = new Linie (new Punkt(rechteck.e1.x,
rechteck.e2.y), new Punkt(rechteck.e2.x,rechteck.e1.y));
   }

   public void zeichne (Graphics g)
   {
     //Aufruf der Methode "zeichne" für das entsprechende Objekt
     rechteck.zeichne(g);
     linie1.zeichne(g);
     linie2.zeichne(g);
   }

   //Variable
   Rechteck rechteck;
   Linie linie1, linie2;
}
```

Letztendlich wird das Grafiksystem selbst implementiert. Es verwendet zur Verwaltung aller grafischen Objekte eine einfache Liste (Typ Vector), welche beliebig erweitert werden kann. Die Speicherverwaltung obliegt vollständig JVM.

```
import java.applet.*;//für Applet
import java.awt.*;   //für Graphics
import java.util.*;     //für Vector

public class Grafiksystem extends Applet
{

   public void paint(Graphics g)
   {
      //Neuzeichnen aller Elemente
      for (int i=0; i<elemente.size(); i++)
      {
         //Referenz auf das Element
         Grafikelement element = (Grafikelement)
elemente.elementAt(i);
         //Aufruf der Methode "zeichne"
         element.zeichne(g);
      }
   }

   public void hinzufuegen(Grafikelement e)
   {
      elemente.addElement(e);
   }

   public void loeschen (Grafikelement e)
   {
      elemente.removeElement(e);
   }

   //Variable
   Vector elemente = new Vector();
}
```

Da noch keine weitere Benutzerinteraktion vereinbart wird erfolgt das Testen dieses 1. Prototyps durch eine abgeleitete Klasse mit einer einfachen Funktion „test“, die innerhalb des Konstruktors einmal aufgerufen wird und die erforderlichen grafischen Elemente vereinbart.

```
//Test der Klasse Grafiksystem
import java.awt.*;

public class GrafiksystemTest extends Grafiksystem
{
  //Konstruktor
  public GrafiksystemTest()
  {
    //einige Daten zum Testen
    test();
  }

  private void test()
  {
    //Nur zum Testen
    //verschiedene Möglichkeiten, Objekte zu instanziieren
    Punkt a = new Punkt(); a.x=100;
    Punkt b = new Punkt(100,100);
    Linie l1 = new Linie(a,b); hinzufuegen(l1);
    Linie l2 = new Linie(b, new Punkt(0,200)); hinzufuegen(l2);
    hinzufuegen (new Linie(b, new Punkt(200,200)));

    Rechteck r1 = new Rechteck(b, new Punkt(300,300));
hinzufuegen(r1);

    Kreis k1 = new Kreis (b, 100); hinzufuegen(k1);
    hinzufuegen(new Kreis (b, 200));

    Rechteck r2 = new Rechteck(new Punkt(30,30), new
Punkt(90,90));
    Kasten ka = new Kasten (r2); hinzufuegen(ka);
  }
}
```

Der Vollständigkeit halber hier noch die erforderliche HTML-Seite zum Starten des Applets.

```
<html>
<head>
<title>GrafiksystemTest</title>
</head>
<body>
<hr>
<applet
  code=Grafiksystem
  width=400
  height=400>
</applet>
<hr>
</body>
</html>
```

5 Benutzeroberflächen

Die Zeiten der dialogorientierten Programme sind gezählt. Keine moderne nichttriviale Anwendung kommt ohne grafische Benutzeroberflächen aus. Während es in herkömmlichen Sprachen oft sehr mühselig war, diese zu entwicklen (von der Verfügbarkeit auf verschiedenen Plattformen ganz zu schweigen) ist dies unter Java mit Hilfe des AWT (Abstract Window Toolkit) sehr einfach möglich.

5.1 Abstract Window Toolkit (AWT)

Das AWT bietet alle gebräuchlichen Formen von Klassen zur Gestaltung beliebiger Oberflächen. Die wichtigsten davon sind in Abbildung 5-1 zusammengestellt. Die *kursiv* dargestellten Klassen sind nicht instanziierbar. Sie dienen lediglich der Vererbung.

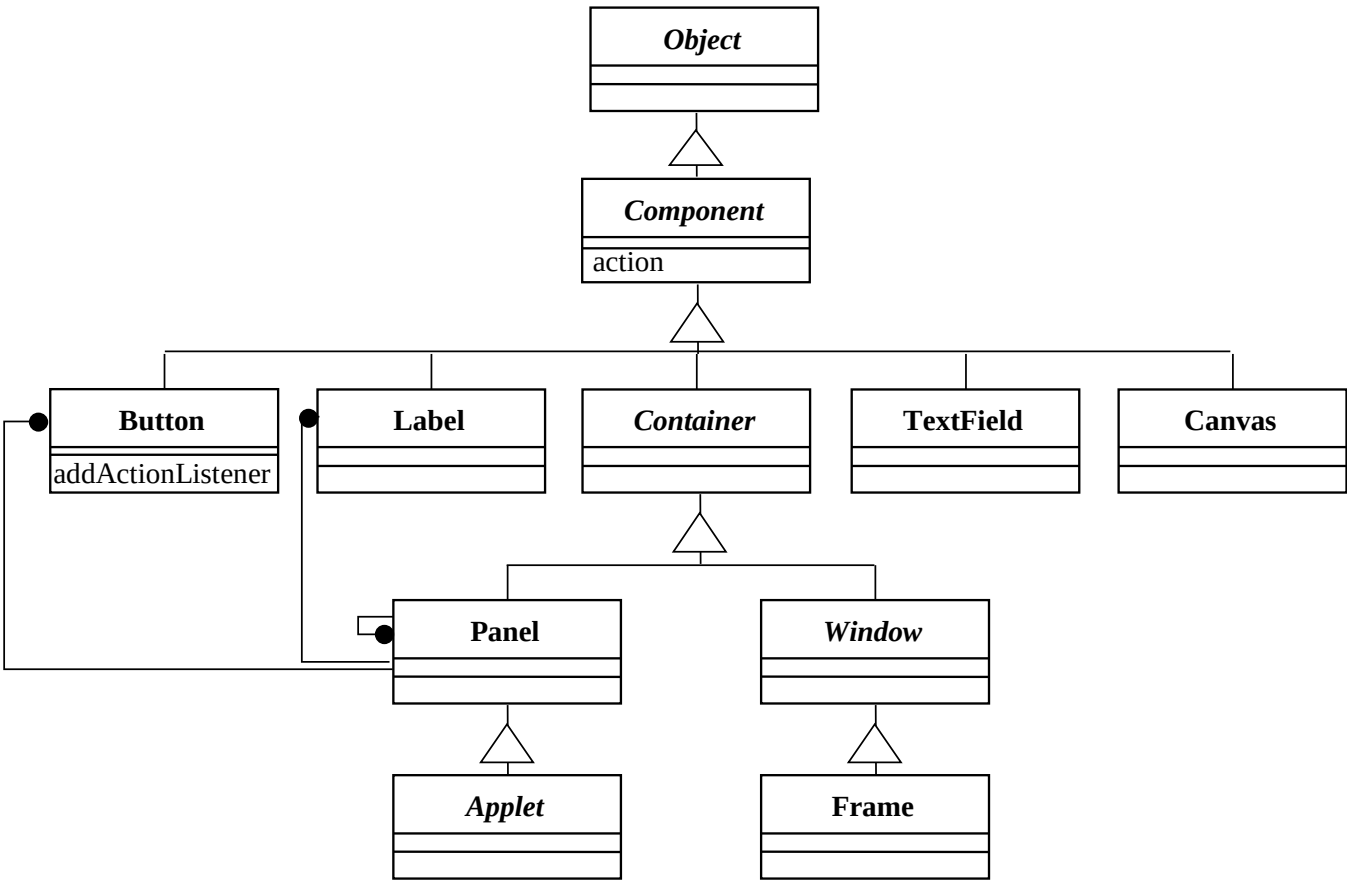


Abbildung 5-1 Ausschnit Klassenhierarchie „AWT“

Die „primitiven“ Elemente mit denen der Benutzer tatsächlich interagiert sind die Elemente in der Mitte von Abbildung 5-1 (Button etc.). Die Anordnung dieser Elemente erfolgt innerhalb eines Sammelobjektes (Containers) der Klasse „Panel“.

Eine Besonderheit stellt die Tatsache dar, daß ein Applet stets von einem Panel abgeleitet ist. Dies bedeutet, daß innerhalb eines Java-Applets auch direkt Objekte zur Benutzeroberfläche abgelegt werden können. Dies sollte jedoch als Sonderfall angesehen werden. Für eine bessere Kontrolle der Objekte bzw. zum besseren Layout sollten stets eigene Panels definiert werden. Ein Panel kann dabei auch weitere Panels beinhalten.

5.1.1 Objekthierarchie vs. Klassenhierarchie

Durch den strukturierten Aufbau von Benutzeroberflächen (Objekte beinhalten weitere Objekte) kommt es zu einer weiteren Hierarchie, der Objekthierarchie. Bei der Objekthierarchie stehen Objekte in einem Vater-Kind-Abhängigkeitsverhältnis im Gegensatz zur Klassenhierarchie, bei der Klassen in einem Vererbungsverhältnis stehen. Objekthierarchien sind stets Baumstrukturen (ein Ast besitzt keine, eine oder mehrere Verzweigungen), ausgehend von einer gemeinsamen Wurzel. Am Beispiel eines Buttons in einer einfachen Benutzeroberfläche soll dies verdeutlicht werden. Der Button steht in der Klassenhierachie aus Abbildung 5-1. Er erbt also die Eigenschaften von „Component“ und „Object“. Die Objekthierarchie eines speziellen Buttons (also ein erzeugtes bzw. instanziiertes Objekt) könnte etwa wie in Abbildung 5-2 aussehen.

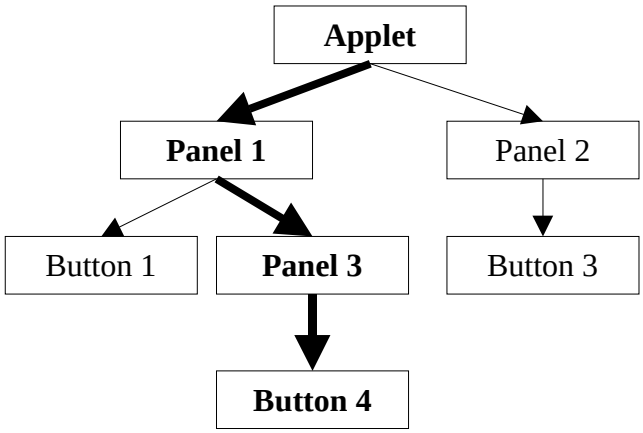


Abbildung 5-2 Objekthierarchie

Die Vater-Kind-Beziehungen (parent-child-relations) auf dem Weg zum Objekt „Button 4“ sind durch die dickeren Pfeile gekennzeichnet. So ist zum Beispiel „Button 4“ nur dann

sichtbar, wenn auch alle Objekte oberhalb der Hierarchie sichtbar sind. Werden Aktionen auf „Button 4“ ausgeführt (vgl. 5.1.3) und von diesem Objekt nicht weiter behandelt, so können die Objekte auf dem Hierarchiepfad (in entsprechend umgekehrter Reihenfolge) diese Aktionen beantworten.

5.1.2 Objekte erzeugen

Als Beispiel soll dem in 4.7.1.3 implementierten Grafiksystem ein Button hinzugefügt werden. Hier kann man auch schön die Möglichkeit der Vererbung in objektorientierten Programmiersprachen demonstrieren. Die bisherige Implementierung der Klasse „Grafiksystem“ soll unverändert bleiben. Die Erweiterung erfolgt nun einfach dadurch, daß die alte Klasse „Grafiksystem“ als Basisklasse benutzt wird und in der neuen Klasse lediglich die weiteren Eigenschaften definiert werden.

```
import java.awt.*;

public class GrafiksystemButton extends Grafiksystem
{
   public GrafiksystemButton()
   {
      //Definition des Buttons
      Button b = new Button("Test");
      //Hinzufügen des Buttons zum Panel
      add(b);
   }
}
```

5.1.3 Benutzerinteraktion

Eine Benutzeroberfläche macht natürlich nur dann Sinn, wenn auch Aktionen eines Benutzers (Ereignisse, Events) erfolgen können und diese dann innerhalb der Anwendung abgearbeitet werden. Dies erfolgt in Java durch vollständige Integration in das objektorientierte Modell einfach dadurch, daß die Klasse „Container“ eine Methode „action“ zur Verfügung stellt, welche bei Interaktion mit diesem Objekt aufgerufen wird.

Unsere letzte Implementierung des Buttons soll nun auch eine Aktion erhalten. Dabei soll nach Drücken des Buttons eine neue horizontale Linie (0,50) bis (150,50) gezeichnet werden. Die Definition der mit dieser Aktion verbundenen „action“-Methode kann auf 3 Arten erfolgen:

- AWT leitet automatisch Ereignisse, die bei einem Objekt eintreffen und nicht über eigene action-Methoden bearbeitet werden zum darunterligenden Objekt (i.d.R. ein Panel) weiter. Das bedeutet, daß innerhalb des Panels alle Ereignisse zu Objekten innerhalb dieses Panels abgearbeitet werden können. Dies ist die gebräuchlichste Technik.

```
import java.awt.*;

public class GrafiksystemButton1 extends GrafiksystemButton
{
   public boolean action(Event  e, Object  o)
   {
      //Definition einer neuen Linie
      Linie l = new Linie(new Punkt(0,50), new Punkt(150,50));
      hinzufügen(l);

      //Neu zeichnen
      repaint();

      return true;
   }
}
```

- Die Behandlung von Aktionen kann in AWT auch delegiert werden. Alle Objekte, welche Aktionen empfangen können besitzen die Methode „addAction Listener“. Jedes Objekt einer Klasse, welche die Schnittstelle „ActionListener“ implementiert, kann dieses Ereignis empfangen und verarbeiten.

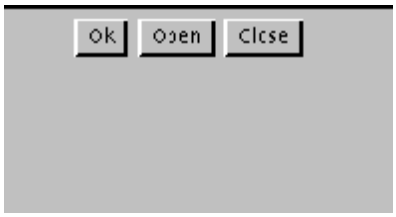
```
import java.awt.*;
import java.awt.event.*;

public class GrafiksystemButton2 extends Grafiksystem
implements ActionListener
{
   public GrafiksystemButton2()
   {
     //Definition des Buttons
     Button b = new Button("Test 2");
     //Hinzufügen des Buttons zum Panel
     add(b);

     //Hinzufügen zum ActionListener
     b.addActionListener(this);
   }

   public void actionPerformed(ActionEvent e)
   {
     //Definition einer neuen Linie
     Linie l = new Linie(new Punkt(0,50), new Punkt(150,50));
     hinzufügen(l);

     //Neu zeichnen
     repaint();
   }
}
```

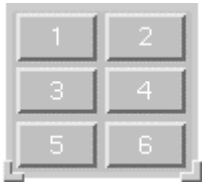
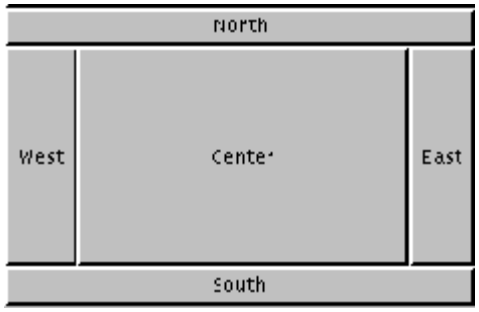
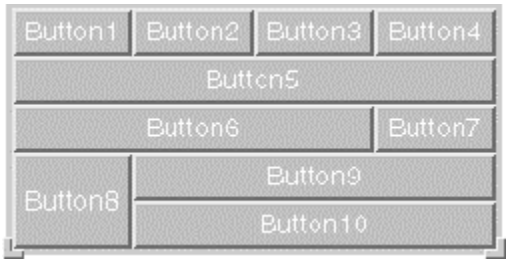
- Es wird von der benötigten Klasse (z.B. Button) eine neue Klasse abgeleitet (z.B. LinienButton). In dieser Klasse wird die Methode „action“ implementiert.
  Der Nachteil dieser Technik ist, daß quasi für jede Instanz eine eigene Klasse definiert werden muß, was sehr schnell zu großen Klassenhierarchien führt, bei denen meistens nur eine Instanz je Klasse existiert. Dies kann aber sinnvoll sein, wenn solche Interaktionsobjekte an verschiedenen Stellen einer Anwendung mehrfach genutzt werden sollen.

```
import java.awt.*;

public class GrafiksystemButton3 extends Grafiksystem
{
   public GrafiksystemButton3()
   {
      //Definition des Buttons
      LinienButton b = new LinienButton(this);
      //Hinzufügen des Buttons zum Panel
      add(b);
   }
}

class LinienButton extends Button
{
   public LinienButton(Grafiksystem system)
   {
      super("Test 3");
      this.system = system;
   }

   public boolean action(Event  e, Object  o)
   {
      //Definition einer neuen Linie
      Linie l = new Linie(new Punkt(0,50), new Punkt(150,50));
      system.hinzufügen(l);

      //Neu zeichnen
      system.repaint();

      return true;
   }

   //Verbindung zum Grafiksystem
   Grafiksystem system;
}
```

5.1.4 Layout

Jedes Panel besitzt eine standardmäßige Strategie zum Layout seiner Kinder. Diese Strategie ist jedoch nicht immer sinnvoll. Soll ein bestimmtes Layout erreicht werden, so muß diese Strategie geändert werden. Dies erfolgt durch sogenannte Layoutmanager. Die derzeit im JDK 1.2 vorhandenen Layoutmanager sowie die implementierten Layouts sind:

Layoutmanager	**Art des Layouts**	**Beispiel**
FlowLayout	Standard-Layout alle Komponenten werden von links nach rechts angeordnet	

Layoutmanager	Art des Layouts	Beispiel
GridLayout	Alle Komponenten werden wie bei einer Tabelle in gleich großen Zeilen und Spalten angeordnet.	
BorderLayout	Das Panel wird anlog zu den Himmelsrichtungen in 5 Bereiche eingeteilt, welche jeweils die gesamte Breite bzw. Höhe des Panels umfassen. Der Zentrums-bereich füllt automatisch den restlichen verfügbaren Platz auf.	
GridBagLayout	Ermöglicht sehr flexibles Layout auch einzelner Komponenten, ist allerdings auch in der Programmierung am aufwendigsten.	
CardLayout	Alle Komponenten werden übereinander gelegt, sodaß immer nur eine Komponente gleichzeitig sichtbar ist (analog Property-Sheets von MS-Windows)	

5.1.4.1 Layoutbeispiel

Das in Abbildung 5-3 dargestellte Layout soll implementiert werden.

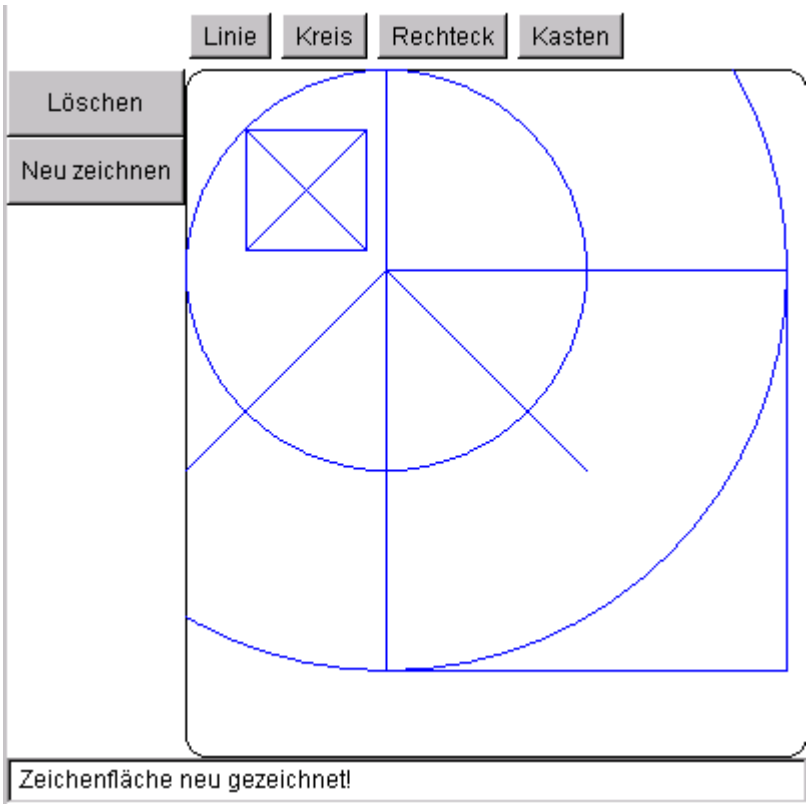


Abbildung 5-3 Layoutbeispiel

Die Objekthierarchie ergibt sich nach Abbildung 5-4

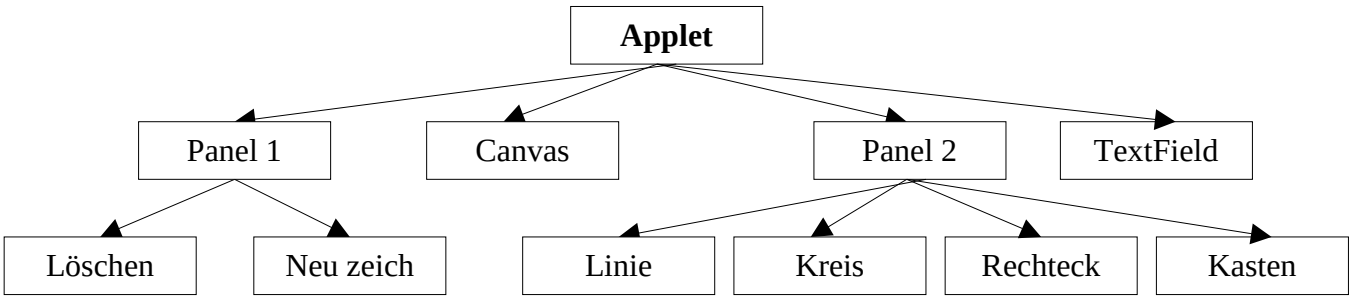


Abbildung 5-4 Layoutbeispiel: Objekthierarchie

Die Layoutmanager ergeben sich zu:

- BorderLayout für Applet
  Die Zeichenfläche (Klasse Canvas) wird in das Zentrum gesetzt und füllt damit den verbleibenden Raum des Applets aus

- GridLayout für Panel 1
  Die Zeilenanzahl kann beliebig variiert werden. Dies beeinflußt die Größe der eingefügten Buttons.

```
import java.awt.*;

class GrafikLayout extends Grafiksystem
{
   public GrafikLayout()
   {
      setzeLayout();
   }

   private void setzeLayout()
   {
      //neues Layout für Applet-panel
      BorderLayout layout1 = new BorderLayout();
      setLayout(layout1);

      //linke Leiste
      Panel links = new Panel();
      links.setLayout(new GridLayout(10,1));
      add("West",links);

      //obere Leiste
      Panel oben = new Panel();
      add("North",oben);

      //Statuszeile
      status = new TextField();
      add("South",status);

      //Rest der Fläche wird zur Zeichenfläche
      zeichenflaeche = new Zeichenflaeche(this);
      add("Center",zeichenflaeche);

      //Einfügen aller Buttons
      loeschen = new Button("Löschen"); links.add(loeschen);
      neu = new Button("Neu zeichnen"); links.add(neu);

      linie = new Button("Linie"); oben.add(linie);
      kreis = new Button("Kreis"); oben.add(kreis);
      rechteck = new Button("Rechteck"); oben.add(rechteck);
      kasten = new Button("Kasten"); oben.add(kasten);
   }

   //Oberflächenelemente
   protected Zeichenflaeche zeichenflaeche;
   protected TextField status;

   protected Button loeschen, neu, linie, kreis, rechteck,
kasten;
}

class Zeichenflaeche extends Canvas
{
   public Zeichenflaeche (Grafiksystem system)
   {
      this.system = system;
   }

   public void paint(Graphics g)
   {
      loeschen();
      //aktuelle Vordergrundfarbe
      g.setColor(Color.blue);
      //Zeichnen der Objekte übernimmt Basisklasse
      system.paint(g);
   }

   public void loeschen()
   {
      //aktuelle Größe
      Dimension dim = size();
      Graphics g = getGraphics();
```

```
      g.clearRect(0,0,dim.width-1,dim.height-1);
      //Rahmen zeichnen
      g.setColor(Color.black);
      g.drawRoundRect(0,0,dim.width-1,dim.height-1, 20,20);
   }

   Grafiksystem system;
}
```

Einige Aktionen sollen nun implementiert werden. Auch hier wird dafür eine neue Klasse abgeleitet und die Methoden zur Ereignisbehandlung definiert.

```
import java.awt.*;

public class GrafikLayoutActions extends GrafikLayout
{
   public boolean action(Event event, Object  o)
   {
      //Aktion abhängig vom Objekt
      if (event.target == loeschen)
      {
         //zeichenfläche löschen
         zeichenflaeche.loeschen();
         status.setText("Zeichenfläche gelöscht!");
      }
      else if (event.target == neu)
      {
         zeichenflaeche.repaint();
         status.setText("Zeichenfläche neu gezeichnet!");
      }
      else if (event.target == linie)
      {
         //Definition einer neuen Linie
         Linie l = new Linie(new Punkt(0,50), new Punkt(150,50));
         hinzufuegen(l);

         //Neu zeichnen
         zeichenflaeche.repaint();
         status.setText("Neue Linie hinzugefügt!");
      }

      return true;
   }

   public boolean mouseDown (Event e, int x, int y)
   {
      Point p = zeichenflaeche.location();
      x-=p.x; y-=p.y;
      status.setText("Mausklick bei (" + x + "," + y + ")");
      return true;
   }
}
```

6 Parallele Prozesse (Threads)

6.1 Allgemeines

Herkömmliche Programme laufen stets sequentiell ab. Ausgehend vom Start des Programms läßt sich zu jeder Zeit während des Programmablaufs eine Stelle als aktuell bearbeitete Anweisung eindeutig identifizieren. Diese Art des Programmablaufs bringt jedoch für einige Anwendungen Probleme, wie z.B.

- Geschwindigkeitsoptimierung durch parallel ablaufende Algorithmen, d.h. Ausnutzen von Mehrprozessorarchitekturen
- dynamische Objekte, d.h. Objekte welche ein Verhalten implementieren, welches sich zeitabhängig verändert ohne daß eine direkte Abhängigkeit zu anderen Objekten besteht.
- Client-Server-Implementierungen, bei denen ein Server mehrere Clients mit gleichen Anforderungen parallel bedienen muß. Die Abarbeitung eines Clients darf die anderen Clients nicht behindern.

Solche Problemstellungen lassen sich in Java sehr einfach durch den Einsatz von Threads lösen. Ein Thread ist dabei kein vollständiger Prozeß, da er im gleichen Adressraum wie der erzeugende Prozeß abläuft (und wird daher in der Literatur auch als „leightweight process“ bezeichnet). Der Thread kann alle Objekte des laufenden Prozesses mitbenutzen, hat aber seine eigene Ablaufkontrolle.

6.2 Implementierung von Threads

Java stellt 2 Möglichkeiten der Implementierung von Threads zur Verfügung.

- Klasse „Thread“
  Alle Objekte einer von dieser Klasse abgeleiteten Klasse können als Threads ablaufen. Dazu muß lediglich die Methode „run“ definiert werden als Einstiegspunkt für den Ablauf des Threads.

- Interface „Runnable“

  Durch die Implementierung der Methode „run“ wird auch hier der Einstiegspunkt in den Thread definiert. Zuvor muß allerdings noch ein Objekt der Klasse „Thread“ mit diesem Objekt als Argument instanziiert werden.

An einem einfachen Beispiel sollen die beiden Möglichkeiten erläutert werden. Eine Klasse soll definiert werden, welche die aktuelle Uhrzeit in einem einstellbaren Aktualisierungsintervall in einem Fenster ausgibt. Die Klasse zur Ermittlung und Darstellung der Uhrzeit ist recht einfach:

```
import java.awt.*;
import java.util.*;

public class Uhr
{
   public Uhr (Panel panel)
   {
      zeit = new TextField();
      setzeZeit();
      panel.add(zeit);
   }

   public void setzeZeit()
   {
      Date datum = new Date();
      zeit.setText("Datum: " + datum);
   }

   //Variable
   TextField zeit;
}
```

Die Uhr wird innerhalb eines Panels (z.B. Applet) dargestellt. Die Änderung der Uhrzeit erfolgt durch Aufruf der Methode „setzeZeit“. Das entsprechende Applet lautet wie folgt:

```
import java.awt.*;
import java.applet.*;

public class TestThread extends Applet
{
   public TestThread ()
   {
      uhr1 = new Uhr(this);
      uhr2 = new Uhr(this);

      add (new Button("Aktuelle Zeit"));
   }

   public boolean action(Event e, Object o)
   {
      uhr1.setzeZeit();
      uhr2.setzeZeit();

      return true;
   }

   Uhr uhr1, uhr2;
}
```

Die Aktualisierung der Uhr erfolgt hier durch den Benutzer (Drücken des Buttons). Das Objekt (uhr1, uhr2) besitzt keine Möglichkeit einer eigenständigen Korrektur.

6.3 Implementierung über Klasse „Thread“

Durch Ableitung von der Klasse „Thread“ und durch Definition der Methode „run“ wird die Möglichkeit eines eigenständigen Threads geschaffen.

```
import java.awt.*;
import java.util.*;

public class UhrThread extends Thread
{
   public UhrThread (Panel panel, int intervall)
   {
      this.intervall = intervall;

      zeit = new TextField();
      setzeZeit();
      panel.add(zeit);
   }

   public void run()
   {
      //Endlosschleife
      while (true)
      {
         setzeZeit();

         try
         {
            Thread.sleep(intervall * 1000);
         }
         catch (Exception e)
         {}
      }
   }


   public void setzeZeit()
   {
      Date datum = new Date();
      zeit.setText("Datum: " + datum);
   }

   //Variable
   TextField zeit;
   int intervall;
}
```

Der Thread muß nun nach der Instanziierung nur noch aktiviert werden:

```
      uhr1 = new UhrThread(this, 1); uhr1.start();
      uhr2 = new UhrThread(this, 2); uhr2.start();
```

Damit laufen beide Threads parallel nebeneinander ohne gegenseitige Beieinflussung. Die Threads laufen im gleichen Adressraum, daher können sie auch jederzeit auf die im Ursprungsprozess definierten Objekte und Methoden zugreifen.

6.4 Implementierung über Schnittstelle „Runnable“

Diese Art der Implementierung kommt insbesondere dann in Betracht, wenn die Methode der Vererbung nicht benutzt werden kann (z.B. weil schon eine Vererbung benutzt wird und Mehrfachvererbung in Java nicht möglich ist). Lediglich die Deklaration der Klasse ändert sich hierbei. Die Methode „run“ bleibt unverändert.

```
public class UhrRunnable implements Runnable
{
   public UhrRunnable (Panel panel, int intervall)
```

Allerdings muß nach der Instanziierung eines solchen Objektes erst noch ein zusätzliches Thread-Objekt erzeugt werden.

```
uhr1 = new UhrRunnable(this, 1); Thread t1 = new Thread(uhr1);
t1.start();
uhr2 = new UhrRunnable(this, 10); Thread t2 = new Thread(uhr2);
t2.start();
```

6.5 Implementierung dynamischer Objekte

Am Beispiel unseres Grafiksystems sollen nun dynamische Objekte definiert werden. Dazu soll eine Klasse „Auto“ entwickelt werden, welches sich in vorgebbarer Geschwindigkeit über die Zeichenfläche bewegt. Die Kontrolle liegt vollständig innerhalb der Klasse, daher wird diese als Thread implementiert. Die Implementierung der Grafikfunktionalität erfolgt analog zu 5.1.4.1. Zur Threadimplementierung wird die Klasse von Thread abgeleitet. Die Geschwindigkeit wird in pixel/s angegeben und die Häufigkeit der Darstellung errechnet sich aus der Forderung, daß das Auto dargestellt wird sobald es die Strecke von 1 Pixel bewältigt hat.

```
import java.awt.*;

public class Auto extends Thread implements Grafikelement
{
  public Auto (Punkt position, Color farbe, int
geschwindigkeit, Zeichenflaeche zeichenflaeche)
  {
    this.position = position;
    this.farbe = farbe;
    this.geschwindigkeit = geschwindigkeit;
    this.zeichenflaeche = zeichenflaeche;
  }

  public void zeichne (Graphics g)
  {
    if (altePosition != null)
    {
      g.setColor(Color.white);
      zeichneNeu(g, altePosition);
    }

    g.setColor(farbe);
    zeichneNeu(g, position);

    altePosition = new Punkt(position.x, position.y);
  }

  private void zeichneNeu (Graphics g, Punkt position)
  {
    Punkt
      p1 = new Punkt(position.x, position.y-10),
      p2 = new Punkt(position.x+20, position.y),
      p3 = new Punkt(position.x+60, position.y),
      p4 = new Punkt(position.x+80, position.y),
      p5 = new Punkt(p2.x,p1.y-10),
      p6 = new Punkt(p3.x,p1.y);

    Rechteck unten = new Rechteck(p1,p4);
    Kreis rad1 = new Kreis(p2,8);
    Kreis rad2 = new Kreis(p3,8);
    Rechteck oben = new Rechteck(p5,p6);

    unten.zeichne(g);
    oben.zeichne(g);
    rad1.zeichne(g);
    rad2.zeichne(g);
  }

  public void run()
  {
    //Zeichnen nach Strecke "1 pixel"
    int warten = 1000 / geschwindigkeit;  //in ms
    while (true)
    {
      try
      {
        sleep(warten);
        zeichne(zeichenflaeche.getGraphics());
        if (vorwaerts)
          position.x++;
        else
          position.x--;
        if (position.x > 300)
          vorwaerts = false;
        if (position.x < 0)
          vorwaerts = true;
      }
      catch (Exception e) {}
    }
  }
```

```
   Punkt position;
   Color farbe;
   int geschwindigkeit;
   Zeichenflaeche zeichenflaeche;
   private boolean vorwaerts = true;
   private Punkt altePosition;
}
```

Das Applet zum Testen besitzt dann den folgenden Aufbau:

```
import java.awt.*;

public class DynGrafik extends GrafikLayoutActions
{
   public DynGrafik()
   {
     //Definition zweier Autos
     Auto polo = new Auto (new Punkt(0,100), Color.red, 10,
zeichenflaeche);
     Auto mercedes = new Auto (new Punkt(0,300), Color.yellow,
20, zeichenflaeche);

     //Threads starten
     polo.start();
     mercedes.start();

     //weitere Autos
     new Auto (new Punkt(0,150), Color.green, 30,
zeichenflaeche).start();
     new Auto (new Punkt(0,250), Color.blue, 40,
zeichenflaeche).start();
     new Auto (new Punkt(0,350), Color.magenta, 50,
zeichenflaeche).start();
   }
}
```

6.6 Synchronisation zwischen Threads

Bei parallel ablaufenden Prozessen ist mitunder eine Synchronisation zwischen den Prozessen erforderlich, inbesondere bei

- Ausführen von zeitkritischen Passagen

  Solche Passagen sollten nicht von mehreren Threads gleichzeitig durchlaufen werden

- Manipulation von gemeinsam benutzten Objekten (Variable, Datenbank etc.)

  Es muß sichergestellt werden, daß ein Thread nicht gerade eine Variable ändert die sich in Benutzung durch einen anderen Thread befindet.

```
//gemeinsamer Vektor elemente existiert

public void test1()
{
   ...
   elemente.remove(object);
   ...

public void test2()
{
   ....
    elemente.addElement(object);
   ...
}
public int test3()
{
   ...
   return elemente.size();
}
```

Java bietet hier die Möglichkeit, einem Thread exkusive Rechte zur Bearbeitung eines Objektes zu geben. Die Zuteilung von Berechtigungen erfolgt dabei stets auf Objektebene, d.h. ein beliebiges Java-Objekt wird als Schlüsselobjekt (Lock) benutzt. Der Thread, welcher den Schlüssel besitzt darf die entsprechend gekennzeichneten Passagen durchlaufen, andere Threads müssen warten bis der Schlüssel wieder frei wird. Die Kennzeichnung der Passagen samt des darin verwendten Schlüssels erfolgt durch:

synchronized (*objektname***)**
{ //Zuteilung des Schlüssels
//kritische Passage
....
} //Freigabe des Schlüssels

Einzige Bedingung ist, daß das entsprechende Schlüsselobjekt "*objektname*" für alle beteiligten Threads eindeutig ist. Im folgenden Beispiel:

```
public void test()
{
   //Beginn kritische Passage
   synchronized (this)
   {
      ...
   }
}
```

wird das "this"-Objekt, also das diese Funktion selbst aufrufende Objekt als Schlüsselobjekt benutzt. Da aber nun jeder Thread ein eigenes Objekt ist (mit natürlich verschiedenem this), bekommt jeder Thread seinen eigenen Schlüssel und damit erfolgt keine Synchronisation. Damit gibt es zur Verwendung von Schlüsselobjekten 2 Möglichkeiten:

- Definition globaler Objekte

  Damit können Schlüssel vergeben werden, welche über eine gesamte Anwendung hinweg zur Synchronisation dienen können.

```
class Schluessel
{
   public static Object
      schluessel1 = new Object(),
      schluessel2 = new Object();
}
```

```
//und irgendwo in einer Klasse
public void test()
{
   synchronized (Schluessel.schluessel1)
   {
      ...
   }
}
```

  Objekt "Schluessel.schlüssel1" ist innerhalb der ganzen Anwendung eindeutig, d.h. es kann nur einen Thread geben, der zu einer bestimmten Zeit die Funktion test1 durchläuft.

- Schlüssel auf Klassenebene

  Damit werden alle kritischen Passsagen innerhalb einer Klasse geschützt. Es gibt jedoch die Möglichkeit, daß Passagen in verschiedenen Klassen zeitgleich ausgeführt werden.

```
public void test()
{
   synchronized (getClass())
   {
      ...
   }
}
```

  Die Memberfunktion "getClass()" liefert ein eindeutiges (also stets das gleiche) Objekt zur Repräsentierung der aktuellen Klasse zurück. Dieses ist die allgemein gebräuchliche Methode, da zeitkritische Passagen häufig in einer gemeinsamen Klasse (z.B. Datenbankzugriffe) implementiert sind.

7 Datenbankzugriff

Java unterstützt den Zugriff auf beliebige relationale Datenbanken über eine einheitliche Programmierschnittstelle (API) an (Einige Hersteller bieten bereits auch Packages für objektorientierte Datenbanken an, jedoch ohne einheitliche API). Der direkte Zugriff auf die Datenbank erfolgt dabei vollständig über SQL (Standardized Query Language). Anhang 9.2 zeigt die gebräuchlichsten SQL-Befehle.

Der Zugriff auf eine Datenbank erfolgt über 3 Schritte:

- Definition des Zugriffsmechanismus (Laden des Java-Datenbank-Treibers)

```
//Registrierung des Treibers
Class.forName(driver);
```

- Verbindung zur Datenbank

```
//Verbindungsaufbau
DriverManager.getConnection (name, "Benutzer", "Passwort");
```

- Formulierung einer Abfrage

```
   //Abfrage
   ResultSet doQuery(String sql)
   {
      try
      {
        //Erzeuge statement
        Statement stm = connection.createStatement();
        ResultSet rs = stm.executeQuery (sql);
        return rs;
      }
      catch (Exception e)
      {
        return null;
      }
   }
```

- Auswertung der Abfrage

  Dies stellt die eigentliche Verbindung zwischen den Datenbankobjekten und Java-Objekten her. Da die Datenbank nur relational arbeitet (d.h. Tabellenstruktur anstatt Klassenstruktur), müssen die einzelnen Werte aus den Tabellen den Java-Objekten zugewiesen werden. Dies ist tatsächlich auch die einzige Schnittstelle zwischen den Daten in der Anwendung und der Datenbank.

```
   private Vector resultsToLinie(ResultSet results)
   {
      if (results == null)
         return null;

      //Alle Resultate in Vektor ablegen
      Vector linien = new Vector();

      try
      {
         // Schleife über alle Resultate
         while (results.next() )
         {
            // Jedes Datenelemnt lesen
            Punkt anfang = new Punkt();
            Punkt ende = new Punkt();

            //Einfache Tabellenabfrage über Spaltennamen
            anfang.x = ((Integer)
results.getObject("xa")).intValue();
            anfang.y = ((Integer)
results.getObject("ya")).intValue();
            ende.x = ((Integer)
results.getObject("xe")).intValue();
            ende.y = ((Integer)
results.getObject("ye")).intValue();

            Linie l = new Linie(anfang, ende);
            linien.addElement(l);
         }
         results.close();

      }
      catch (Exception e)
      {
         System.out.println("Ausnahme: " + e);
      }

      //Vektor auf exakte Größe anpassen
      linien.trimToSize();
      return linien;
   }
```

7.1 Sinnvolle Klassenhierarchie

Abbildung 7-1 zeigt eine mögliche Klassenhierarchie zur Realisierung einer Datenbankanbindung.

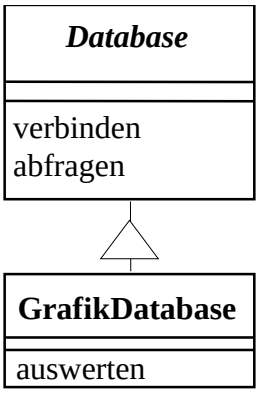


Abbildung 7-1 Klassenhierachie Datenbank

Eine abstrakte Klasse "Database" stellt alle von der eigentlichen Anwendung unabhängigen Funktionen zur Verfügung. Die Anbindung an die anwednungsspezifischen Datenstrukturen erfolgt in einer speziell für diese Anwendung implementierten Klasse "GrafikDatabase".

7.1.1 Klasse "Database"

```
//Implementierung einer ODBC-Database

import java.sql.*;
import java.util.*;
//Für Microsoft ODBC
import com.ms.*;


public abstract class Database
{
  public Database(String name, String user, String password)
throws Exception
  {
    try
    {
      //Registrierung des Treibers
      Class.forName(driver);
      //Verbindungsaufbau
      connection = DriverManager.getConnection (name,
"Benutzer", "Passwort");
    }

    catch (Exception e)
    {
      throw e;
    }

  }

  //Abfrage
  ResultSet doQuery(String sql)
  {
    try
    {
      //Erzeuge statement
      Statement stm = connection.createStatement();
      ResultSet rs = stm.executeQuery (sql);
      return rs;
    }
    catch (Exception e)
    {
      return null;
    }
  }

  //Aktualisierung
  int doUpdate(String sql)
  {
    //Updates synchronisieren
    synchronized (this.getClass())
    {
      try
      {
        //Create statement
        Statement stm = connection.createStatement();
        int n = stm.executeUpdate (sql);
        stm.close();
        return n;
      }
      catch (SQLException e)
      {
        return 0;
      }
    } //end sync
  }

  //Umwandlung Java-Timestamp nach SQL-String
  String timeToString(Timestamp ts)
```

```
   {
     //convert to valid SQL time string
     StringBuffer buff = new StringBuffer();
     buff.append("{ ts '");
     buff.append(ts.toString());
     //remove nanoseconds
     buff.setLength(buff.length()-2);
     buff.append("'}");

     return buff.toString();
   }



   //Treibername (je nach Java-Version :-( )
   private String driver = "com.ms.jdbc.odbc.JdbcOdbcDriver";
   //für MS-jview
   //private String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
   //für Sun-java

   private Connection connection;
}
```

7.1.2 Klasse "GrafikDatabase"

Es wird hier die Anbindung an das vorher implementierte Grafiksystem realisiert, zunächst nur mit der Unterstützung für einfache Linienobjekte, d.h. es wird die Fähigkeit implementiert, von einer Datenbank alle Linienobjekte zu laden und Java-Objekte zu erzeugen.

```
//Implementierung einer ODBC-Database

import java.sql.*;
import java.util.*;

public class GrafikDatabase extends Database
{
  //Konstruktor
  public GrafikDatabase() throws Exception
  {
    //Aufruf Database-Konstruktor
    super("jdbc:odbc:GrafikDaten","","");
  }

  public Vector abfrage()
  {
    //Formulierung einer Abfrage
    String query = "SELECT * FROM Linien";
    ResultSet results = doQuery(query);

    Vector linien = resultsToLinie(results);

    return linien;
  }


  //Auslesen einer Datenbankabfrage
  private Vector resultsToLinie(ResultSet results)
  {
    if (results == null)
      return null;

    //Alle Resultate in Vektor ablegen
    Vector linien = new Vector();

    try
    {
      // Schleife über alle Resultate
      while (results.next() )
      {
        // Jedes Datenelement lesen
        Punkt anfang = new Punkt();
        Punkt ende = new Punkt();

        //Einfache Tabellenabfrage über Spaltennamen
        anfang.x = ((Integer)
results.getObject("xa")).intValue();
        anfang.y = ((Integer)
results.getObject("ya")).intValue();
        ende.x = ((Integer)
results.getObject("xe")).intValue();
        ende.y = ((Integer)
results.getObject("ye")).intValue();

        Linie l = new Linie(anfang, ende);
        linien.addElement(l);
      }
      results.close();

    }
    catch (Exception e)
    {
      System.out.println("Ausnahme: " + e);
    }

    //Vektor auf exakte Größe anpassen
    linien.trimToSize();
    return linien;
  }
}
```

Diese Art der Datenbankanbindung ist recht einfach, da ein Java-Objekt (in diesem Fall eine Linie) stellts als vollständiges Objekt innerhalb einer Tabelle in der Datenbank definiert ist, d.h. alle Informationen zu diesem Objekt liegen als einfache Daten (hier Typ Integer) vor.

xa	ya	xe	ye
0	0	100	100
100	100	200	100

Abbildung 7-2 Tabelle "Linie"

Dies wird jedoch bei Verwendung von Assoziationen etwas schwieriger. Nach Abbildung 4-6 verfügt nämlich ein Linienobjekt über eine Assoziation zu einem Punktobjekt. Für die relationale Datenbank bedeutet dies, daß eine weitere Tabelle (z.B. "Punkte") existiert und die Tabelle der Linienobjekte nur Verweise auf die entsprechenden Zeilen der Tabelle Punkte enthält.

nummer	x	y
1	0	0
2	100	100
3	200	100

anfang	ende
1	2
2	3

Abbildung 7-3 Tabellen "Punkte" und "Linien"

Die Verbindung wird hier über eine zugeordnete ID "nummer" hergestellt. Die Realisierung dieser Assoziation innerhalb Java kann über 2 Möglichkeiten erfolgen:

- Getrennte Abfrage nach "Punkte" und "Linien"
  Hier werden alle Punkte gelesen und innerhalb einer Datenstruktur (bevorzugt "Hashtable") abgelegt. Nach dem Abfrage der Linien wird über die gelesene Nummer die entsprechende Koordinate aus der Punkte-Datenstruktur erhalten und entsprechend umgesetzt.
  Der Nachteil dieser Methode ist, daß alle Punkte gelesen werden müssen unabhängig davon, ob diese später auch in entsprechenden Linienobjekten verwendet werden.
  Die Alternative, zunächst die Tabelle "Linien" zu lesen und dann über gezielte Abfragen nur die einzelnen Punkte zu erhalten scheitert an Performanceproblemen bei großen Datenbanken.

```
   public Vector abfrage3()
   {
     //Lesen aller Punkte
     String query = "SELECT * FROM Punkt";
     ResultSet results = doQuery(query);

     Hashtable punkte = resultsToPunkte(results);

     //Lesen aller Linien
     query = "SELECT * FROM Linie";
     results = doQuery(query);

     Vector linien = resultsToLinien(results, punkte);

     return linien;
   }

   private Hashtable resultsToPunkte(ResultSet results)
   {
     if (results == null)
       return null;

     //Alle Resultate in Vektor ablegen
     Hashtable punkte = new Hashtable();

     try
     {
       // Schleife über alle Resultate
       while (results.next() )
       {
         // Jedes Datenelemnt lesen
         Punkt p = new Punkt();

         Integer nummer = (Integer)
results.getObject("nummer");
         p.x = ((Integer) results.getObject("x")).intValue();
         p.y = ((Integer) results.getObject("y")).intValue();

         punkte.put(nummer,p);

       }
       results.close();

     }
     catch (Exception e)
     {
       System.out.println("Ausnahme: " + e);
     }

     return punkte;
   }


   private Vector resultsToLinien( ResultSet results, Hashtable
punkte)
   {
     if (results == null)
       return null;

     //Alle Resultate in Vektor ablegen
     Vector linien = new Vector();

     try
     {
       // Schleife über alle Resultate
       while (results.next())
       {

         //Lesen der Indizes
         Integer anfang = (Integer)
results.getObject("anfang");
```

```
          Integer ende = (Integer) results.getObject("ende");

          //Ermittlung der zugehörigen Punkte
          //Bei nicht vorhandem Punkt wird Exception ausgelöst
          Punkt a = (Punkt) punkte.get(anfang);
          Punkt b = (Punkt) punkte.get(ende);

          Linie l = new Linie(a, b);
          linien.addElement(l);
        }
        results.close();

      }
      catch (Exception e)
      {
        System.out.println("Ausnahme: " + e);
      }

      //Vektor auf exakte Größe anpassen
      linien.trimToSize();
      return linien;
    }
```

- Die Datenbank stellt eine entsprechende Auswahlabfrage zur Verfügung

  Dabei handelt es sich um eine innerhalb der Datenbank definierte Tabelle, welche die Verknüpfung zwischen Objekten berücksichtigt. Eine solche Auswahlabfrage ist in Abbildung 7-4 am Beispiel einer Access-Datenbank abgebildet. Nach der Abfrage wird ein Resultat "Linie" mit den 4 Koordinaten $(x,y)_{Anfangspunkt}$ und $(x,y)_{Endpunkt}$ zurückgeliefert. Eine solche Abfrage wird dann wie bei einfachen Tabellen durchgeführt. Allerdings ist die korrekte Namensgebung der Spalten hier schwierig (im Beispiel existieren die Spalten "x" und "y" doppelt). Daher sollte die Adressierung der Spalten über die Spaltennumer erfolgen.

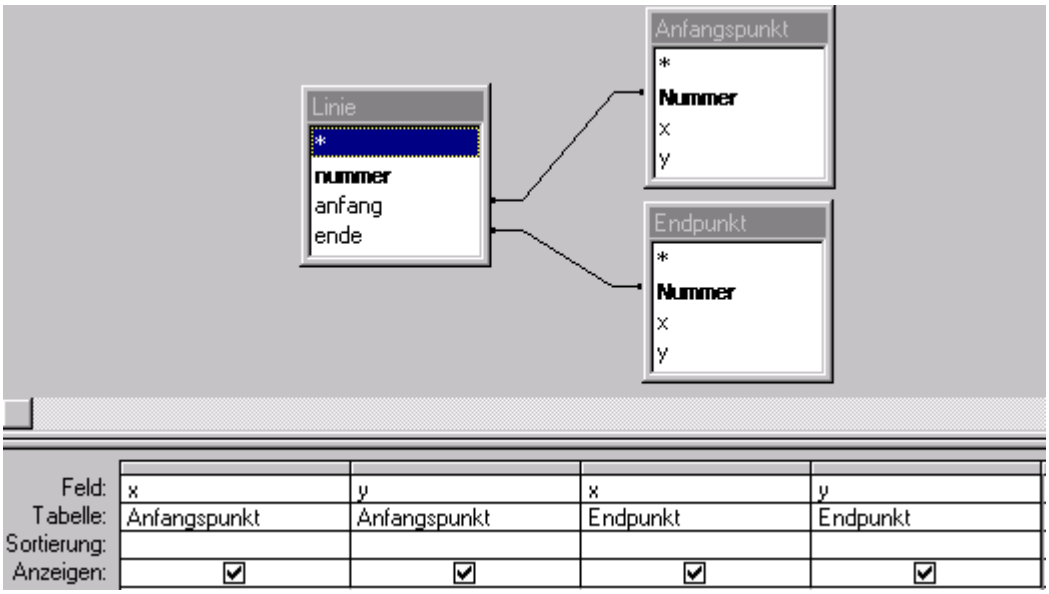


Abbildung 7-4 Auswahlabfrage (MS-Access)

```
   private Vector resultsToLinien(ResultSet results)
   {
     if (results == null)
       return null;

     //Alle Resultate in Vektor ablegen
     Vector linien = new Vector();

     try
     {
       // Schleife über alle Resultate
       while (results.next() )
       {
         // Jedes Datenelemnt lesen
         Punkt anfang = new Punkt();
         Punkt ende = new Punkt();

         //Auswahlabfrage (nur über Spaltennummern!)
         anfang.x = ((Integer)
results.getObject(1)).intValue();
         anfang.y = ((Integer)
results.getObject(2)).intValue();
         ende.x = ((Integer) results.getObject(3)).intValue();
         ende.y = ((Integer) results.getObject(4)).intValue();

         Linie l = new Linie(anfang, ende);
         linien.addElement(l);
       }
       results.close();

     }
     catch (Exception e)
     {
       System.out.println("Ausnahme: " + e);
     }

     //Vektor auf exakte Größe anpassen
     linien.trimToSize();
     return linien;
   }
```

8 Verteilte Anwendungen

Verteilte Anwendungen sind immer dort anzutreffen, wo von verschiedenen Anwendungen aus auf gleiche Datenbestände zugegriffen werden muß. Ganz typische Anwendungsgebiete sind hier Zugriffe auf Datenbanken, welche sich auf Servern irgendwo im Netz befinden. Bei der Anwendung von Java sind verteilte Anwendungen besonders wichtig, da aufgrund der Sicherheit Applets kein Zugriff auf das Dateisystem gestattet ist und damit Daten häufig in Datenbanken abgelegt werden.

8.1 Bestandteile einer verteilten Java-Anwendung

Abbildung 8-1 zeigt eine typische Umgebung für eine verteilte Java-Anwendung.

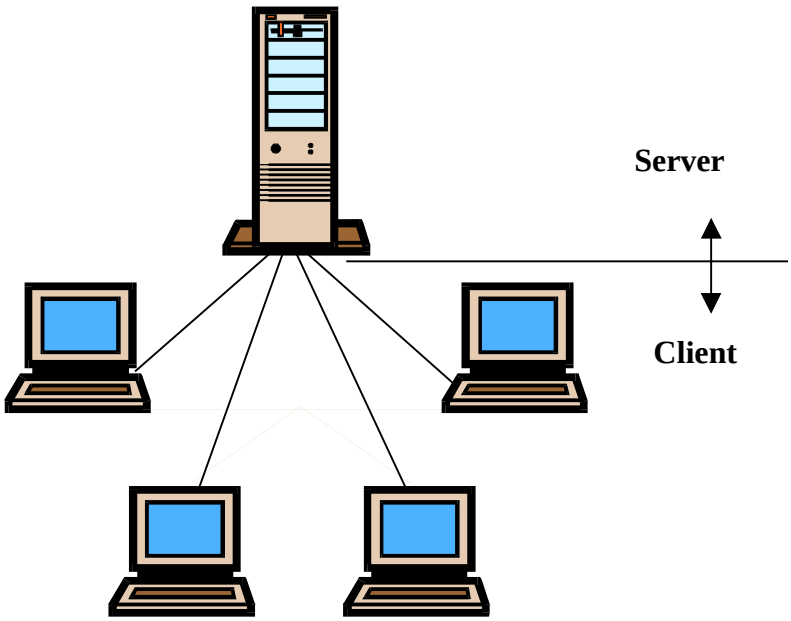


Abbildung 8-1 Client-Server-Anwendung

Beliebig viele Clients greifen auf einen (oder auch mehrere) Server zu und teilen damit stets den gleichen Datenbestand. Im Prinzip ist dies analog zu parallelen Prozessen (vgl. Threads in Kap. 6), nur daß die Clients auf unterschiedlichen Rechnern (und Architekturen) im Netz laufen. Während der Datentransfer bei Threads innerhalb des gleichen virtuellen Adressraums stattfindet erfolgt beim Datenaustausch zwischen Client und Server der Datentransfer über das Netzwerk. Java bietet nun eine direkte Unterstützung (in Form eines Packages) für solche Anwendungen durch das Remote-Method-Interface RMI. Eine Besonderheit bei Java besteht

auch in der Tatsache, daß Clients nicht nur Daten, sondern auch ihren gesamten Code (als Bytecode) vom Server herunterladen.

8.2 Remote Method Interface (RMI)

RMI reduziert die Komplexität einer Datenübertragung im Netzwerk auf den Mechanismus eines Funktionsaufrufes, d.h. ein Funktionsaufruf innerhalb RMI auf Clientseite bewirkt den Aufruf einer entsprechenden Funktion auf Serverseite. Die erforderlichen Funktionsargumente werden dabei über das Netzwerk übertragen und das Resultat des Funktionsaufrufs (Rückgabewert) wiederum zurück an den Client geliefert. Die Komplexität dieses Vorgangs (Netzwerkverbindung, Prozeßkommunikation, Versenden von Daten etc.) wird vollständig innerhalb RMI abgebildet und bleibt dem Anwendungsprogrammierer verborgen. Es ist lediglich ein geringer Verwaltungsaufwand zur Spezifikation der Schnittstellen und der im Netz verwendeten Server erforderlich. Damit ergibt sich die softwareseitige Implementierung nach Abbildung 8-2.

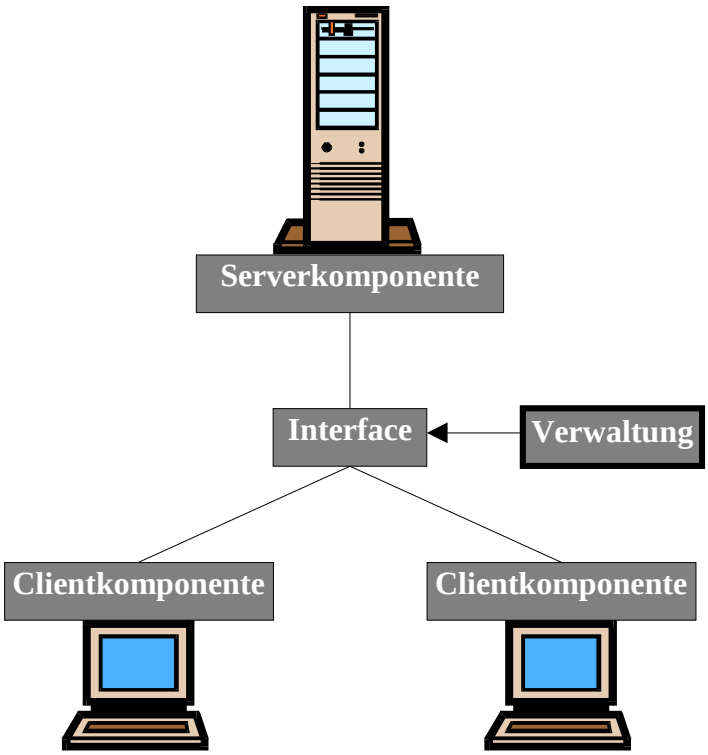


Abbildung 8-2 RMI-Implementierung

8.2.1 Schnittstelle (Interface)

Das Interface legt fest, welche Funktionen zur Kommunikation zwischen Client und Server benutzt werden können. Dies ist ganz analog zu einer "normalen" Java-Schnittstellendefinition (vgl. 4.3.5).

```
//Interface-Definition für Client-Server Grafik
import java.util.*;
import java.rmi.*;

public interface GrafikInterface extends Remote
{
   //Lesen aller Elemente (Typ = GrafikElement)
   public Vector leseElemente () throws RemoteException;
   //Lesen aller Elemente ab bestimmtem Datum
   public Vector leseElemente (Date from) throws
RemoteException;
}
```

Im Beispiel werden 2 Funktionen deklariert, welche zum Lesen von Objekten benutzt werden können. Dier Objekte werden als Vektor zurückgeliefert. Der Funktionsaufruf entspricht dem einer einfachen Java-Funktion. Die Implementierung der Funktion erfolgt stets auf der Serverseite (vgl. 8.2.2). Zur Ausnahmebehandlung (Netzwerkfehler, Server nicht erreichbar etc.) dienen auch hier Exceptions (vgl. 3.9), die von jeder der hier deklarierten Funktionen auch berücksichtigt werden muß.

8.2.2 Serverkomponente

Aufgrund der Sicherheitskonzepte von Java muß die Serverkomponente als Applikation (mit einer "main"-Funktion als Startpunkt) entwickelt werden. Dies ist schon allein wegen der Möglichkeit der Verbindung zu verschiedenen Clients (auf verschiedenen Rechnern) erforderlich. Zur weiteren Festlegung von Sicherheitsrichtlinien muß auch ein spezieller "Security-Manager" initialisiert werden. Dieser ist insbesondere dafür verantwortlich, daß Clients die erforderlichen Interface-Klassen über das Netz laden können. Für normale Anwendungen reicht dafür der Standard-Konstruktor aus. Jeder Server wird über einen eindeutigen Namen gekennzeichnet ("Naming.rebind"). Über diesen Namen wird es den Clients später ermöglicht, Verbindungen aufzubauen.

Innerhalb der Serverkomponente werden alle Methoden aus der Schnittstellendefinition (8.2.1) wie "normale" Memberfunktionen implementiert.

```
//Implementierung GrafikServer

import java.util.*;
//RMI support
import java.rmi.*;
import java.rmi.server.*;

class GrafikServer extends UnicastRemoteObject implements
GrafikInterface
{

   public static void main(String[] args)
   {
      //Security manager
      System.setSecurityManager (new RMISecurityManager());

      //Initialisierung durch Erzeugen eines Objekts
      new GrafikServer();
   }

   public GrafikServer()
   {
      //Registrieren des Servers
      try
      {
         Naming.rebind("/GrafikServer",this);
         System.out.println("Registrierung ok!");
      }
      catch (Exception e)
      {
         //Initialisierungsfehler
         System.out.println("Keine Registrierung möglich!");
      }
   }



   public Vector leseElemente () throws RemoteException
   {
      Vector v = test();
      return v;
   }

   public Vector leseElemente (Date from) throws RemoteException
   {
      //Date wird derzeit nicht unterstützt
      return leseElemente ();
   }

   //zum Testen
   private Vector test()
   {
      Vector v = new Vector();

      //zum Testen: Hinzufügen einiger grafischer Elemente
      Punkt a = new Punkt(); a.x=100;
      Punkt b = new Punkt(100,100);
      Linie l1 = new Linie(a,b); v.addElement (l1);
      Linie l2 = new Linie(b, new Punkt(0,200)); v.addElement
(l2);
      v.addElement (new Linie(b, new Punkt(200,200)));

      Rechteck r1 = new Rechteck(b, new Punkt(300,300));
v.addElement (r1);

      Kreis k1 = new Kreis (b, 100); v.addElement (k1);
      v.addElement (new Kreis (b, 200));

      Rechteck r2 = new Rechteck(new Punkt(30,30), new
Punkt(90,90));
      Kasten ka = new Kasten (r2); v.addElement (ka);
```

```
      //Rückgabe
      return v;
   }
}
```

8.2.3 Clientkomponente

Bei der Implementierung des Clients empfiehlt sich analog zur Datenbankanbindung (vgl. 7.1) eine 2-stufige Vorgehensweise. Zunächst wird eine allgemeine Klasse zur Kommunikation mit dem Server implementiert. Das Auffinden des Servers erfolgt durch die "Naming.lookup"-Funktion. Wichtig dabei ist, daß der Zugriff auf den Server über ein Objekt des Typs "GrafikInterface" (also ein Interface-Objekt, vgl. 8.2.1) und nicht über ein Serverobjekt erfolgt. Der Aufruf der Remote-Memberfunktionen unterscheidet sich nun nicht mehr vom Aufruf lokaler Memberfunktionen (abgesehen von der Ausnahmebehandlung).

```
//GrafikClient
//Verbindet zu GrafikServer und stellt alle erforderlichen
Daten über entsprechende Memberfunktionen zur Verfügung

import java.util.*;
import java.rmi.*;

public class GrafikClient
{
   public GrafikClient(String service)
   {
      //Verbindung zum Server
      try
      {
         //Finden des Servers
         Object o = Naming.lookup(service);
         //Cast auf Interface
         server = (GrafikInterface) o;
      }
      catch (Exception e)
      {
         //Fehler bei Verbindung ☹
      }
   }

   //Lesen der Daten vom Server
   public Vector daten()
   {
      if (server == null)
      {
         //Kein Server vorhanden
         return null;
      }

      try
      {
         return server.leseElemente();
      }
      catch (Exception e)
      {
         //Fehler in Verbindung
         return null;
      }
   }

   private GrafikInterface server;
}
```

In der eigentlichen Implementierung des Clients wird ein Objekt dieser Klasse erzeugt und über dieses Objekt sämtliche Kommunikation abgewickelt. Dabei ist zu beachten, daß aufgrund der Sicherheit Applets nur Verbindungen zum dem Server aufbauen dürfen, von welchem sie auch geladen wurden. Der Servername innerhalb eines Applets kann durch die Applet-Memberfunktion " **getCodeBase().getHost()**" erfragt werden.

```
//Implementierung Client

import java.sql.*;
import java.util.*;

public class TestClientServer extends GrafikLayout
{
   public void init()
   {
     //Abfrage des Servers
     try
     {
       //Erzeugen des Clients
       client = new GrafikClient("//" + getCodeBase().getHost()
+ "/GrafikServer");

       //Lesen der Daten vom Server
       Vector elemente = client.daten();

       if (elemente != null)
       {
         status.setText("Anzahl Daten: " + elemente.size());
         //Hinzufügen allerElemente
         for (int i=0; i<elemente.size(); i++)
           hinzufuegen((Grafikelement) elemente.elementAt(i));
       }
       else
         status.setText("Keine Daten vorhanden: ");

     }
     catch (Exception e)
     {
       //Fehler beim Verbinden zum Server :-(
       status.setText("Fehler in Serververbindung: " + e);
     }
   }

   private GrafikClient client;
}
```

8.2.4 Kommunikation

Damit die Kommunikation zwischen Clients und Server stattfinden kann, sind weitere Vorraussetzungen erforderlich. Da alle Daten beim Aufruf einer Methode zur Client-Serverkommunikation über ein Netzwerk übertragen (und nicht wie bei einer lokalen Methode über den Stack ausgetauscht) werden müssen diese Daten in ein netzwerkfähiges Format übertragen werden. Dies erfolgt in Java einfach dadurch, daß alle Objekte zur Client-Serverkommunikation das Interface "**java.io.Serializable**" implementieren. Die Implementierung der relevanten Funktionen wird dann durch die Basisklasse "Object" vorgenommen. In unserem Beispiel werden Objekte des Typs "Grafikelement" übertragen. Diese wiederum benutzen Objekte des Typs "Punkt". Alle diese Klassen implementieren nun neben dem Interface "Grafikelement" "(vgl. 4.7.1) auch "java.io.Serializable".

```
//"Grafik.java"
//Grafische Elemente

import java.awt.*;

//Punkt
class Punkt implements java.io.Serializable
{
...
}

//Linie
class Linie implements Grafikelement , java.io.Serializable
{
...
}

//Rechteck
class Rechteck implements Grafikelement , java.io.Serializable
{
...
}

//Kreis
class Kreis implements Grafikelement , java.io.Serializable
{
...
}

//Kasten
class Kasten implements Grafikelement , java.io.Serializable
{
...
}
```

Da auf einem Rechner auch mehrere Serverprogramme ablaufen können, ist eine zentrale Stelle erforderlich um die Verbindung des Clients zum jeweiligen Server zu koordinieren. Dabei wird auch vom Client die Implementierung der Schnittstellendefinition vom Server heruntergeladen, da über diese Funktionen dann die gesamte Kommunikation abläuft. Für diesen Zweck gibt es 2 weitere Dienstprogramme als Bestandteil eines jeden Java-Entwicklungssystems.

- Remote Method Interface Compiler (**rmic**)
  Mit diesem Compiler wird das Server-Class-File (in unserem Beispiel "GrafikServer.class") kompiliert und entsprechende Funktionen (sog. "Stubs") erzeugt, die dann später vom Client geladen werden. Syntax ist:
  rmic GrafikServer

- Registrierungstool (**rmiregistry**)
  Dieses Programm muß im gleichen Verzeichnis wie die zugehörigen class-Dateien ausgeführt werden (UNIX: ***rmiregistry&*** bzw. Windows: ***start rmiregistry***). Das Programm wartet (Standardmäßig auf Port 1099) auf ankommende Clients und weist die zugehörigen Stubs zu.

8.3 Zusammenfassung

Abbildung 8-3 zeigt nochmals zusammenfassend die erforderlichen Schritte zur Erstellung einer Client-Server-Umgebung. Die Implementierung von Client und Server kann dabei unabhängig voneinander erfolgen, Basis für beide Entwicklungen ist dabei lediglich die gemeinsame Schnittstellendefinition.

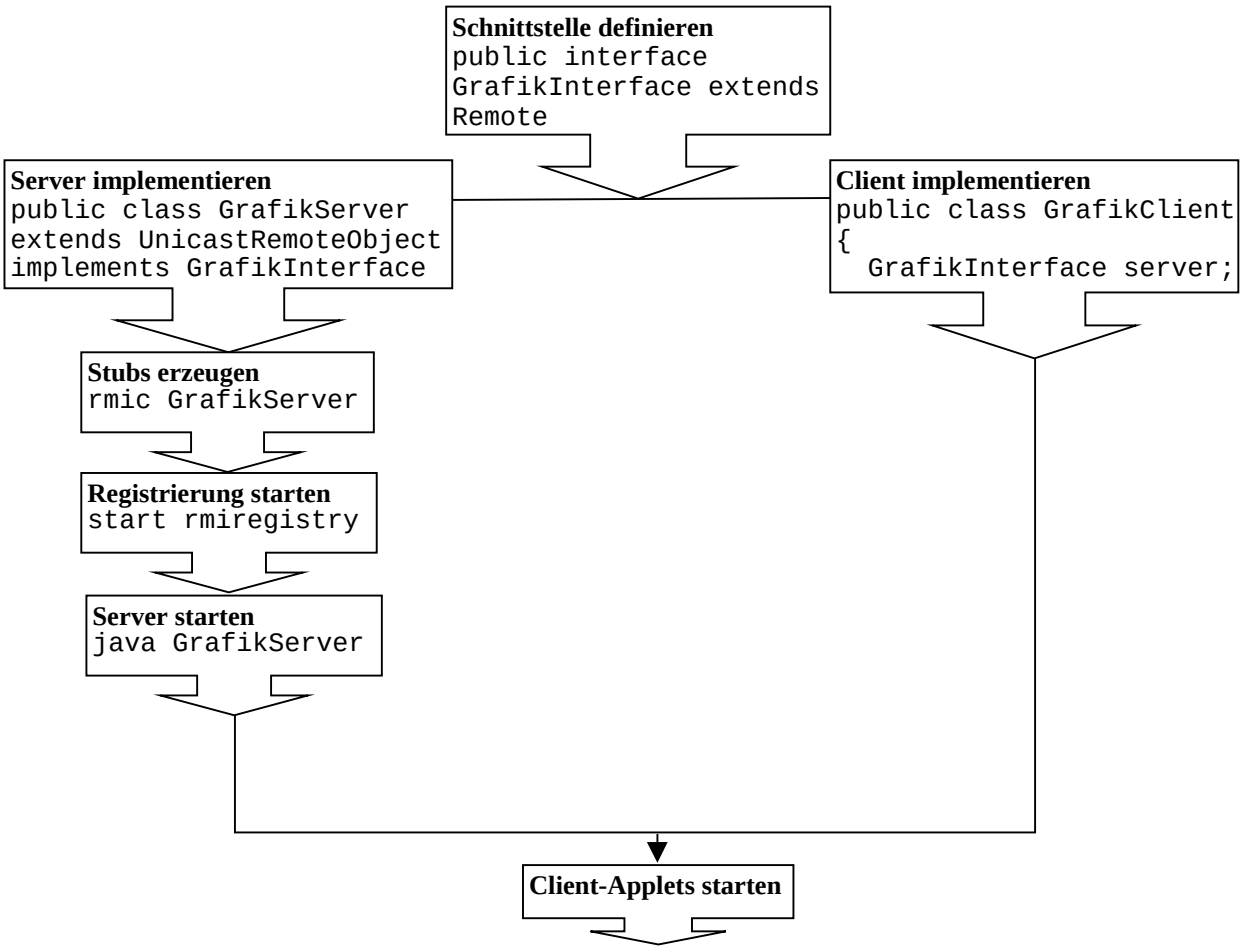


Abbildung 8-3 Programmentwicklung Client-Server

Beim Starten des Applets sind noch die Sicherheitsbestimmungen von Java zu beachten. Da ein Applet nur Verbindungen zu Rechnern aufbauen darf, über welches das Applet auch geladen wurde, kann beim Testen der Applet-Code nicht von der Festplatte geladen werden. Dies ist nur möglich unter Verwendung des "appletviewer". Wird ein "normaler" Browser verwendet, so kann das Laden des Applets nur über einen WWW-Server erfolgen, an dem dann auch das entsprechende Serverprogramm abläuft.

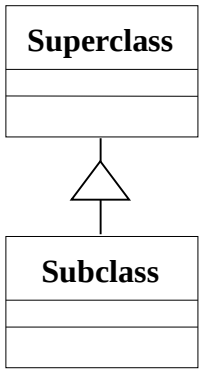
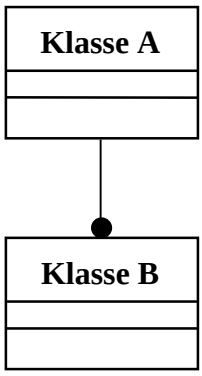
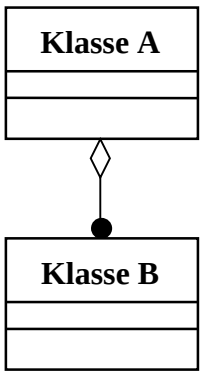
9 Anhang

9.1 Object Modeling Technique (OMT)

Es werden die innerhalb dieses Skripts verwendeten grafischen Notationen zur Modellierung zusammengestellt. Dabei werden diese nur insoweit aufgeführt, als sie auch innerhalb des Skripts angewendet wurden. Zu einer vollständigen Beschreibung der Notation sei auf die Literatur verwiesen.

OMT wurde zuerst von Rumbaugh eingeführt und existiert neben weiteren Techniken zur objektorientierten Modellierung. Sie wurde innerhalb dieses Skripts aufgrund ihrer Möglichkeiten zu einer komprimierten und aussagekräftigen Darstellungsweise des Objektmodells angewendet.

Notation	Beschreibung
	Klasse Notation einer Klasse mit Kennzeichnung des Namens (Klasse), der verwendeten Attribute sowie der anwendbaren Methoden. Die Reihenfolge der Spezifikation ist einzuhalten, Attribute und Methoden können auch weggelassen werden. Attribute sind stets einfache Datentypen. Die Modellierung von Klassenattributen erfolgt stets als Assoziation bzw. Aggregation.
	Vererbung (Generalisierung) Die abgeleitete Klasse „Subclass“ erbt alle Eigenschaften der Basisklasse „Superclass“.

Notation	Beschreibung
	Mehrfachvererbung (d.h. mehrere Basisklassen) ist möglich
	Assoziation Klasse A besitzt 0 oder mehrere Assoziationen zu Klasse B, d.h. Objekte einer Klasse B sind Bestandteil von Objekten der Klasse A. (Innerhalb Java werden zur Implementierung von Assoziationen Referenzen benutzt)
	Aggregation Spezielle Form der Assoziation. Klasse B ist (im Gegensatz zur Assoziation) nicht unabhängig von Klasse A. Dies wird häufig zur Modellierung von Subsystemen (enge Part-of-Beziehung) benutzt.

9.2 Kurzbeschreibung SQL-Syntax

Im Folgenden wird nur ein ganz kurzer Überblick über alle die Funktionen angegeben, welche innerhalb desSkripts und der Beispielprogramme auch verwendet werden. Ausführlichere Beschreibungen zu SQL siehe Literatur bzw. Internet.

Die Anweisungen lassen sich in 2 Gruppen zusammenfassen:

- Auswahl von Werten aus einer Datenbank (Select-Anweisung)
- Schreiben von Werten in eine Datenbank (Update-Anweisung)
- Löschen von Werten (Delete-Anweisung)
- Ändern von Werten (Update-Anweisung)

9.2.1 Allgemeine Syntax

- Befehle kennen keine Unterscheidung zwischen Groß/Keinschreibung
- Zeichenketten sind in '-Zeichen einzuschließen
  Beispiel: SELECT * FROM Pers WHERE Pers.name = 'Hans Meier'
- Datumsangaben sind durch { d 'Datum'} anzugeben
  SELECT * FROM Pers WHERE Pers.geburtsdatum > {d '1950-01-01'}
- Zeitangaben sind durch { ts 'Zeit'} anzugeben
- SELECT * FROM Pers WHERE Pers.arbeitsbeginn > {ts '1998-04-30 9:00:00.000}'
  Die Angabe vom Millisekunden (.000) kann entfallen, allerdings ist die Sekundenangabe obligatorisch

9.2.2 Select-Anweisung

Die Select-Anweisung dient zur Auswahl von Datensätzen aus einer oder mehrerer Tabellen.

SELECT attribute FROM Tabelle WHERE bedingung ORDER BY attribute

Beispiele:

```
SELECT * FROM Linien
SELECT Linien.xa, Linien.ya FROM Linien
SELECT * FROM Linien WHERE Linien.xa > 100
SELECT * FROM Linien WHERE Linien.xa > 0 AND Linien.xa < 100
```

9.2.3 Insert-Anweisung

Die Insert-Anweisung dient zum Hinzufügen von Tabelleneinträgen

INSERT INTO *Tabelle* **(***attribute***) VALUES (***daten***)**

Beispiel

```
INSERT INTO Linien (Linien.anfang,Linien.ende) VALUES (2,4)
```

9.2.4 DELETE-Anweisung

Die Delete-Anweisung dient zum Löschen von Tabelleneinträgen

DELETE FROM *tabelle* **WHERE** *bedingung*

Beispiel

```
DELETE FROM Linie WHERE ende > 5
DELETE FROM Linie where ende = anfang
```

9.2.5 Update-Anweisung

Die Update-Anweisung dient zum Ändern von Tabelleneinträgen.

UPDATE Tabelle SET attribut=wert WHERE bedingung

Beispiel

```
UPDATE Linie SET anfang=5 WHERE nummer = 1
UPDATE Linie SET anfang=ende WHERE nummer = 3
```

9.3 Einrichten einer ODBC-Datenbank unter Windows

Eine einfache Möglichkeit zum Zugriff auf beliebige relationale Datenbanken unter Windows besteht in der Verwendung der Java-ODBC-Bridge. Die ODBC-Treiber sind in Windows enthalten und können direkt unter Java auch benutzt werden. Zuvor muß allerdings noch die ODBC-Datenquelle unter Windows eingerichtet werden. Dies erfolgt in den folgenden Schritten (am Beispiel einer MS-Access Datenbank):

- Datenbank erstellen
  Tabellen anlegen etc.

- ODBC-Datenquelle einrichten
  Systemsteuerung->ODBC

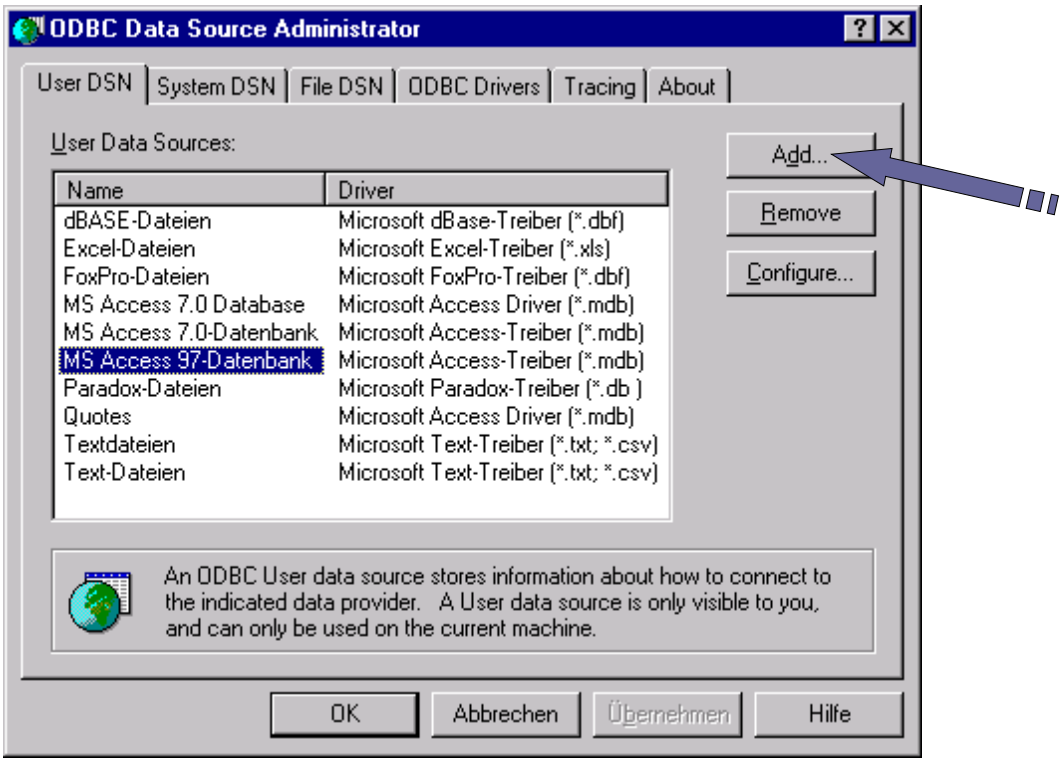


Abbildung 9-1 ODBC-Administration

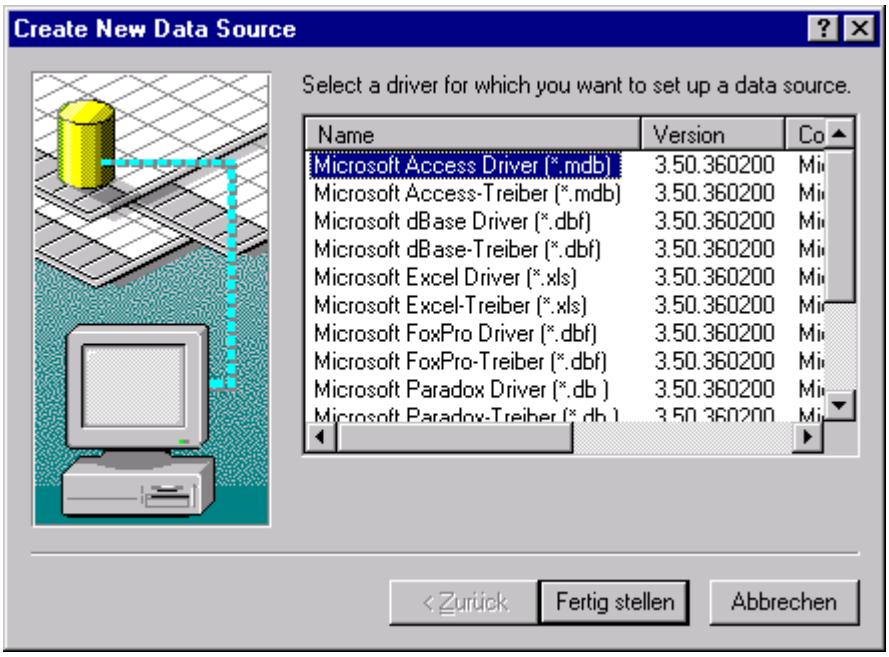


Abbildung 9-2 ODBC-Treiber wählen

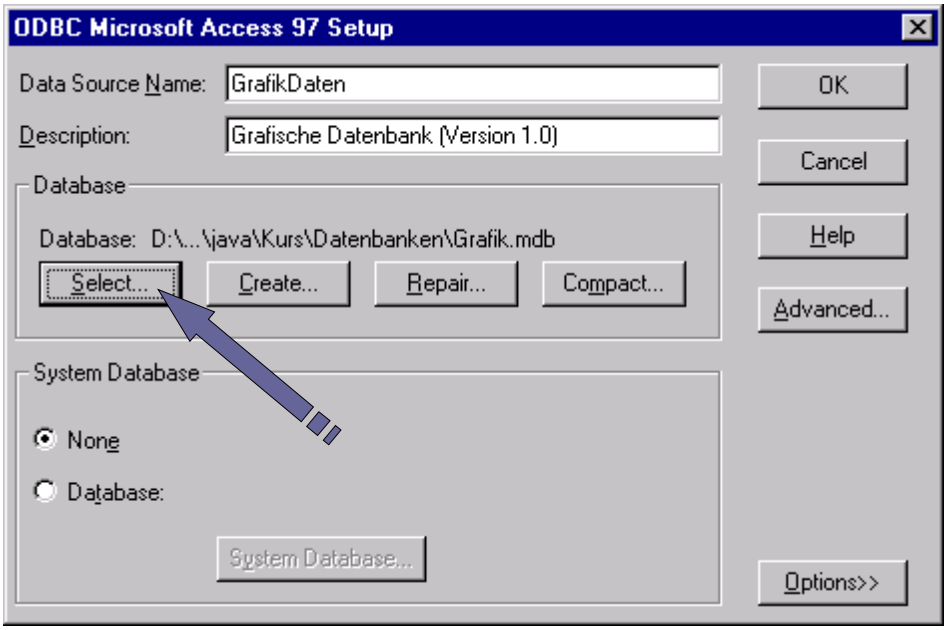


Abbildung 9-3 Datenquelle benennen

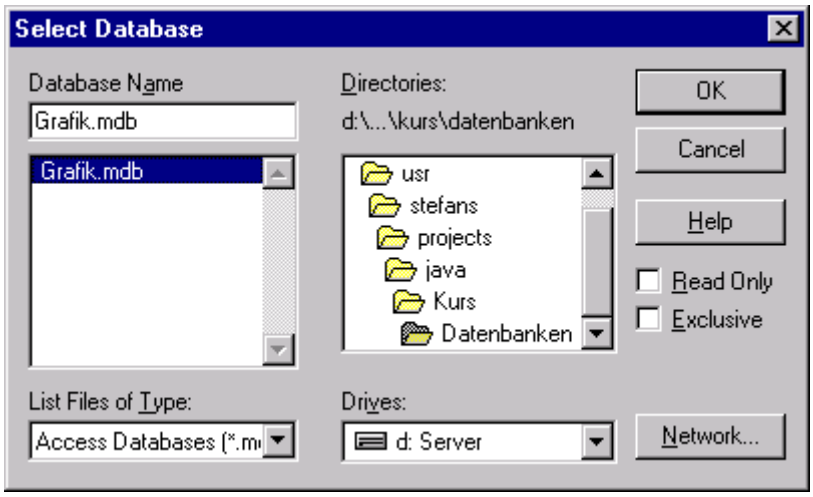
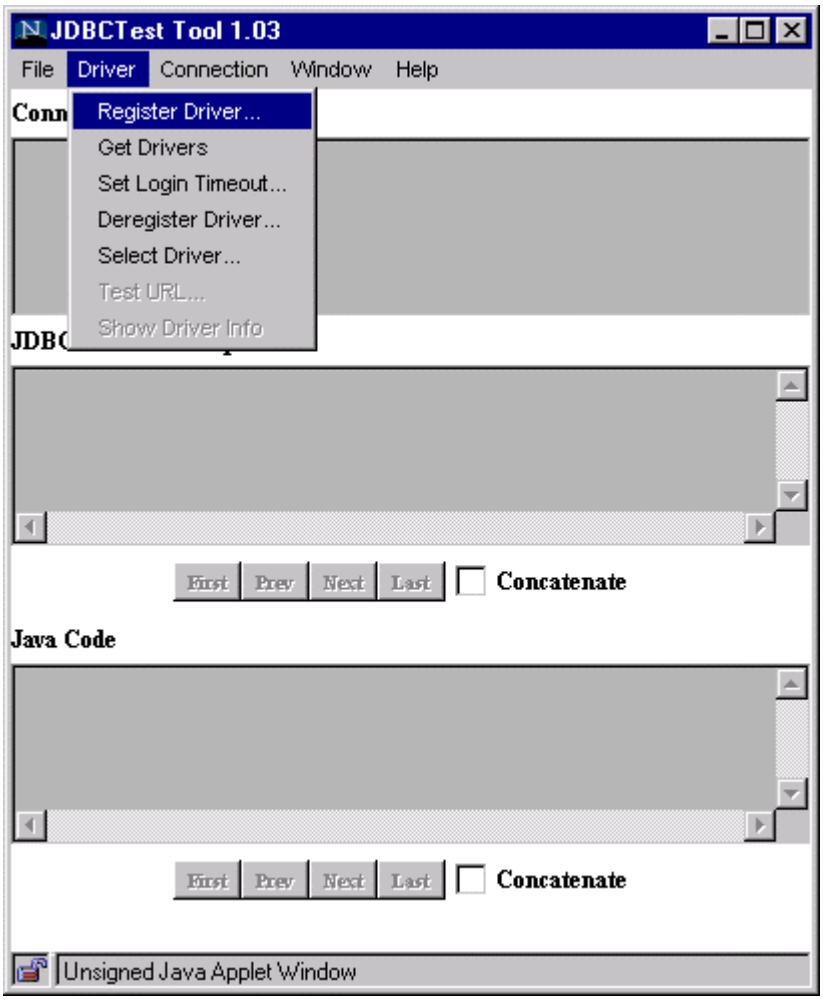


Abbildung 9-4 Datenbankdatei auswählen

- Datenquelle testen

  z.B: http://www.bauv.unibw-muenchen.de/Java/test/JDBCTest1_03/JDBCTest.html



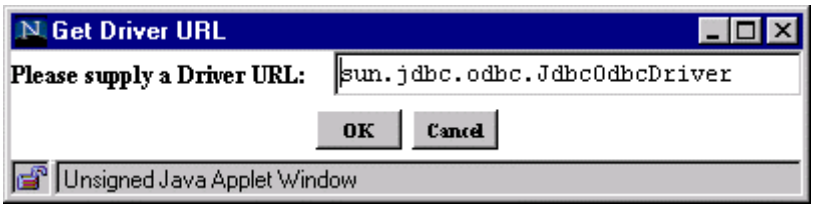


Abbildung 9-5 Treiber registrieren

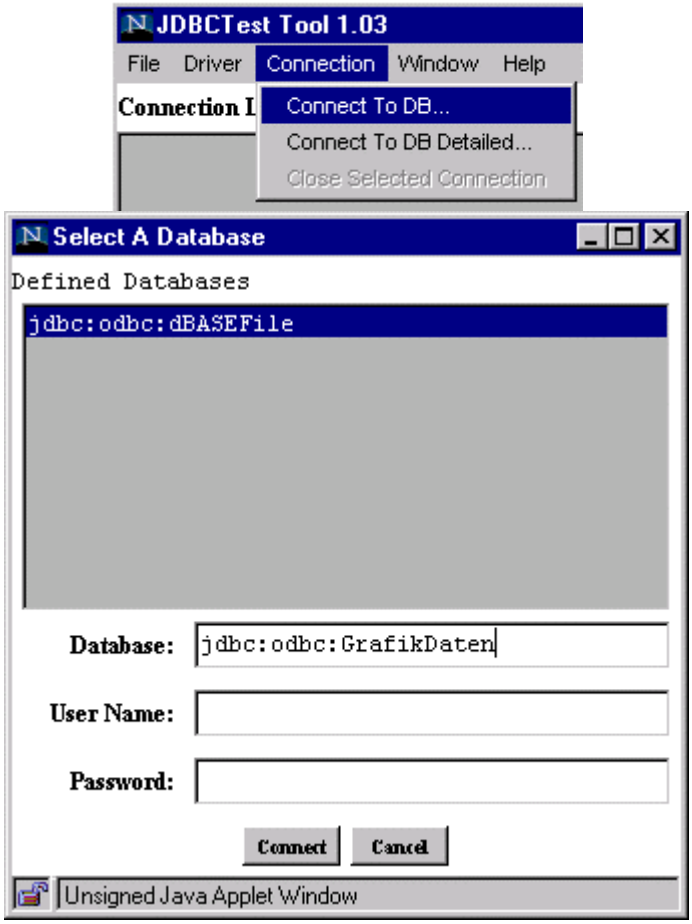


Abbildung 9-6 Verbinden zur Datenquelle

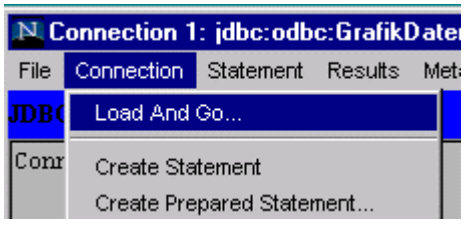


Abbildung 9-7 SQL Statement aktivieren

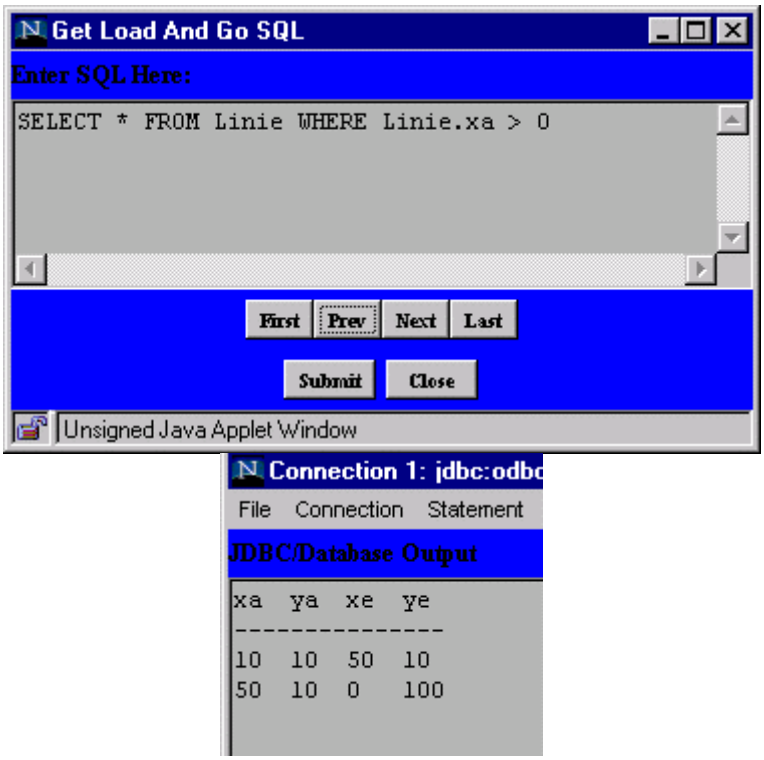


Abbildung 9-8 SQL Anweisung testen

9.4 Arbeiten mit Visual J++

9.4.1 Grundlagen

Es werden die Grundlagen zum Einsatz des Entwicklungssystems „Visual J++“ kurz zusammengestellt. Die Arbeitsweise zum Erstellen einer Java-Anwendung (Applet oder Applikation) gliedern sich dabei in die Schritte:

- Workspace erstellen
  Der Workspace stellt ein Rahmenwerk mit allen globalen Einstellungen für ein aktuelles Projekt zur Verfügung. Er verwaltet zugehörige Dateien, Compiler- und Debug-Einstellungen und stellt fest, wann welche Dateien nach erfolgten Änderungen neu erzeugt werden müssen (entspricht unter UNIX einem „Makefile“ mit zugehörigem „make“-Tool)

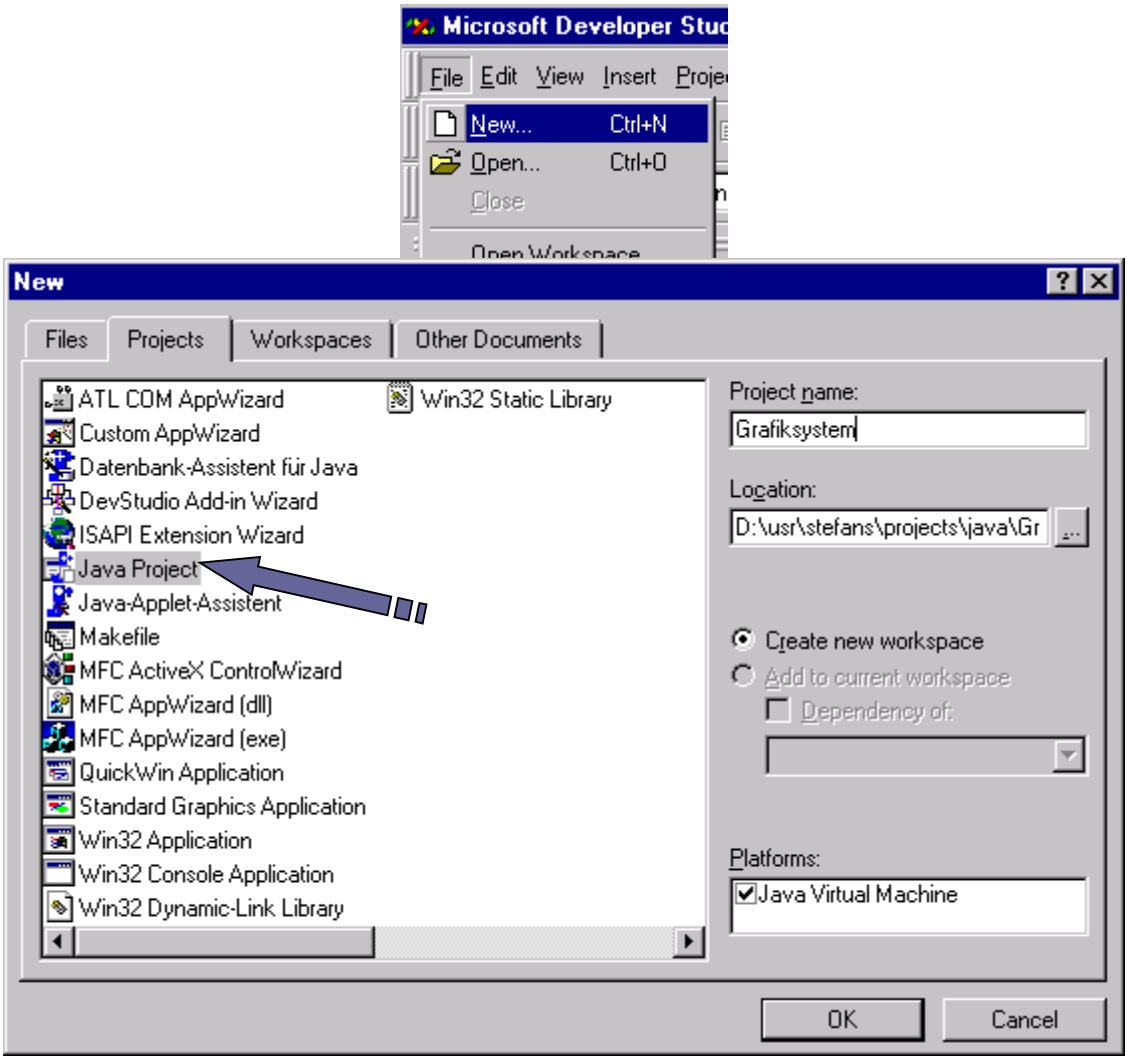


Abbildung 9-9 Workspace erstellen

- Java-Dateien erzeugen

  Eine Java-Datei muß unter der Namenserweiterung „.java“ abgespeichert sein (Menü File->Save As). Dies sollte bei einer neuen Datei sofort erfolgen, da sonst keine Java-Sprachunterstützung (farbige Hervorhebung, automatisches Einrücken etc.) erfolgt.

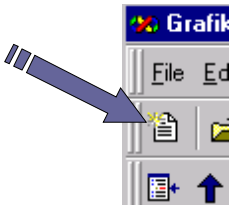
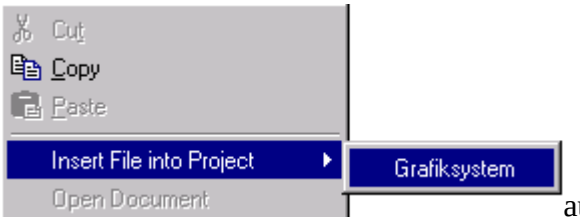


Abbildung 9-10 Neue Java-Datei erzeugen

- Java-Dateien dem Workspace hinzufügen

  Eine Datei wird nur dann übersetzt, wenn sie auch dem aktuellen Workspace angehört. Dazu muß innerhalb des Editors durch Drücken der rechten Maustaste das Menü 

aufgerufen werden. Die Datei wird daraufhin im Workspace angezeigt (Einstellung “FileView”).

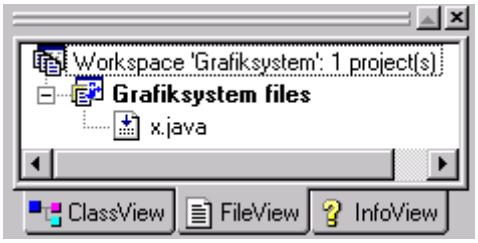


Abbildung 9-11 Java-Datei dem Workspace hinzufügen

- Anwendung übersetzen 

  Alle Dateien innerhalb des aktiven Workspace werden, soweit sie verändert wurden, neu übersetzt.

- Anwendung starten 

  Die Anwendung wird als Applikation direkt bzw. als Applet über einen Browser gestartet. Dazu muß lediglich die Klasse angegeben werden, welche als erste instanziiert werden soll (d.h. bei einer Applikation die Klasse mit der “main”-Methode bzw. beim Applet die Klasse, welche von der Applet-Klasse abgeleitet wurde). Diese Einstellung kann mit Hilfe des Menüs “Project->Settings” jederzeit geändert werden. Standardmäßig ist als Browser MS Internet Explorer eingestellt, jedoch kann unter “Category: Browser” auch ein beliebiger Browser gesetzt werden (Ausnahme: Zum Debuggen ist der Internet Explorer erforderlich).

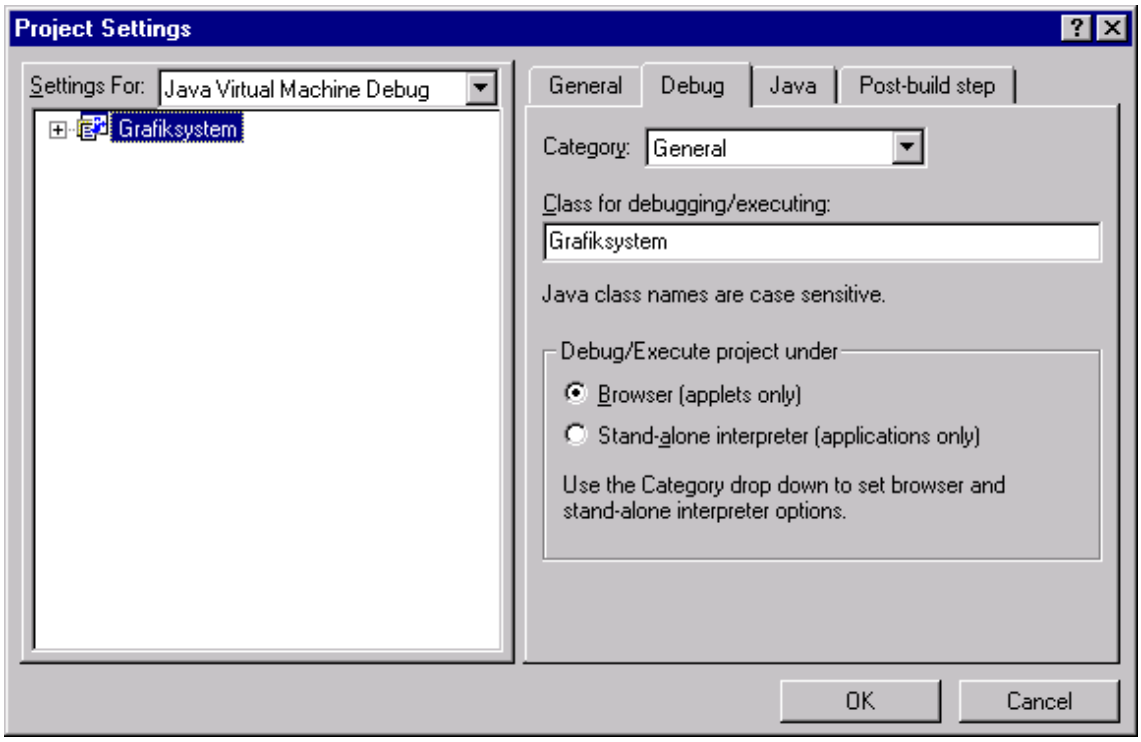


Abbildung 9-12 Einstellung zum Start der Anwendung

- Spezielle HTML-Seite für Applets vereinbaren

  Standardmäßig wird eine HTML-Seite automatisch generiert. Diese ist jedoch in einigen Fällen (z.B. ungeeignete Größe des Applets) nicht vorteilhaft. Daher kann eine beliebige HTML-Seite (mit dem entsprechenden Applet-Tag) eingestellt werden.

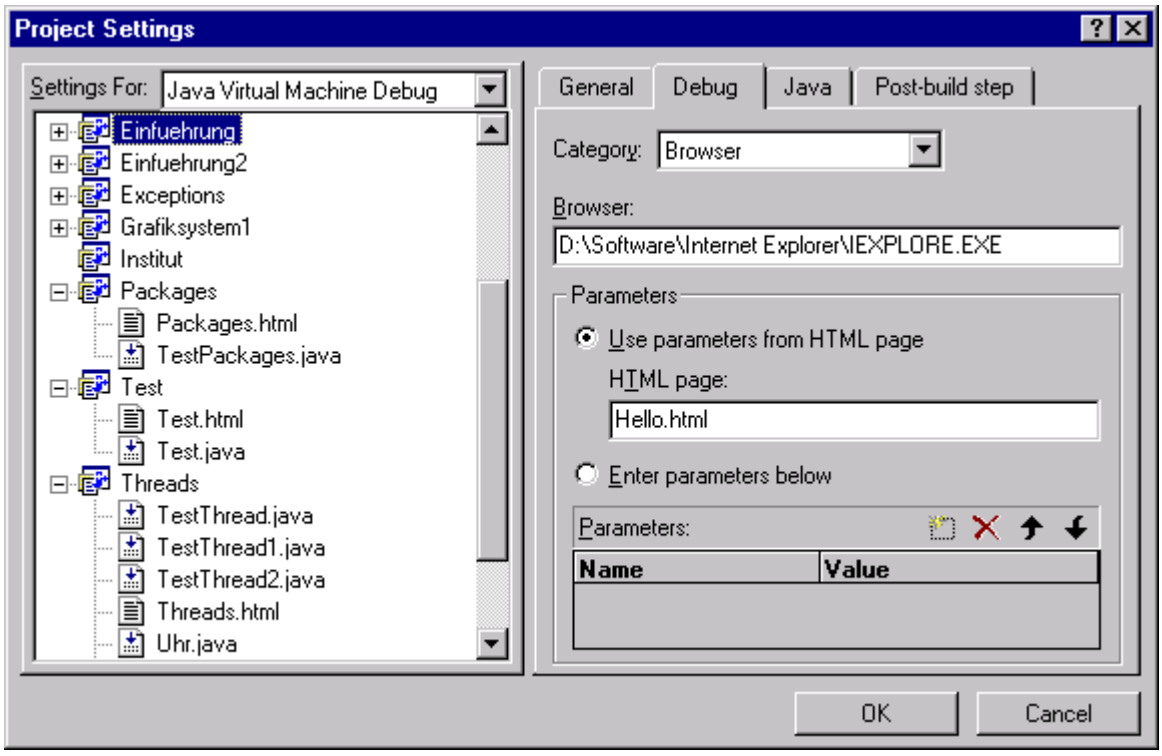


Abbildung 9-13 Eigene HTML-Seite für Applet vereinbaren

- Debuggen (Ablaufkontrolle, Fehlersuche) 
  Zuvor müssen innerhalb des Editors an den interessierenden Stellen Haltepunkte (mind. einer)  gesetzt werden.

9.4.2 Arbeiten mit mehreren Projekten

Dies ist insbeondere dann sinnvoll, wenn Prototypen bzw. Weiterentwicklungen erstellt werden sollen welche bereits vorhandene Klassen mitbenutzen. Solche Prototypen sollten stets in einem eigenen Workspace entwickelt werden. Zur Vereinfachung bietet J++ die Möglichkeit, innerhalb eines globalen Workspace mehrere Subworkspaces zu vereinen. Das Hinzufügen eines neuen Workspace erfolgt durch Drücken der rechten Maustaste über einem bereits vorhandenen Workspace.

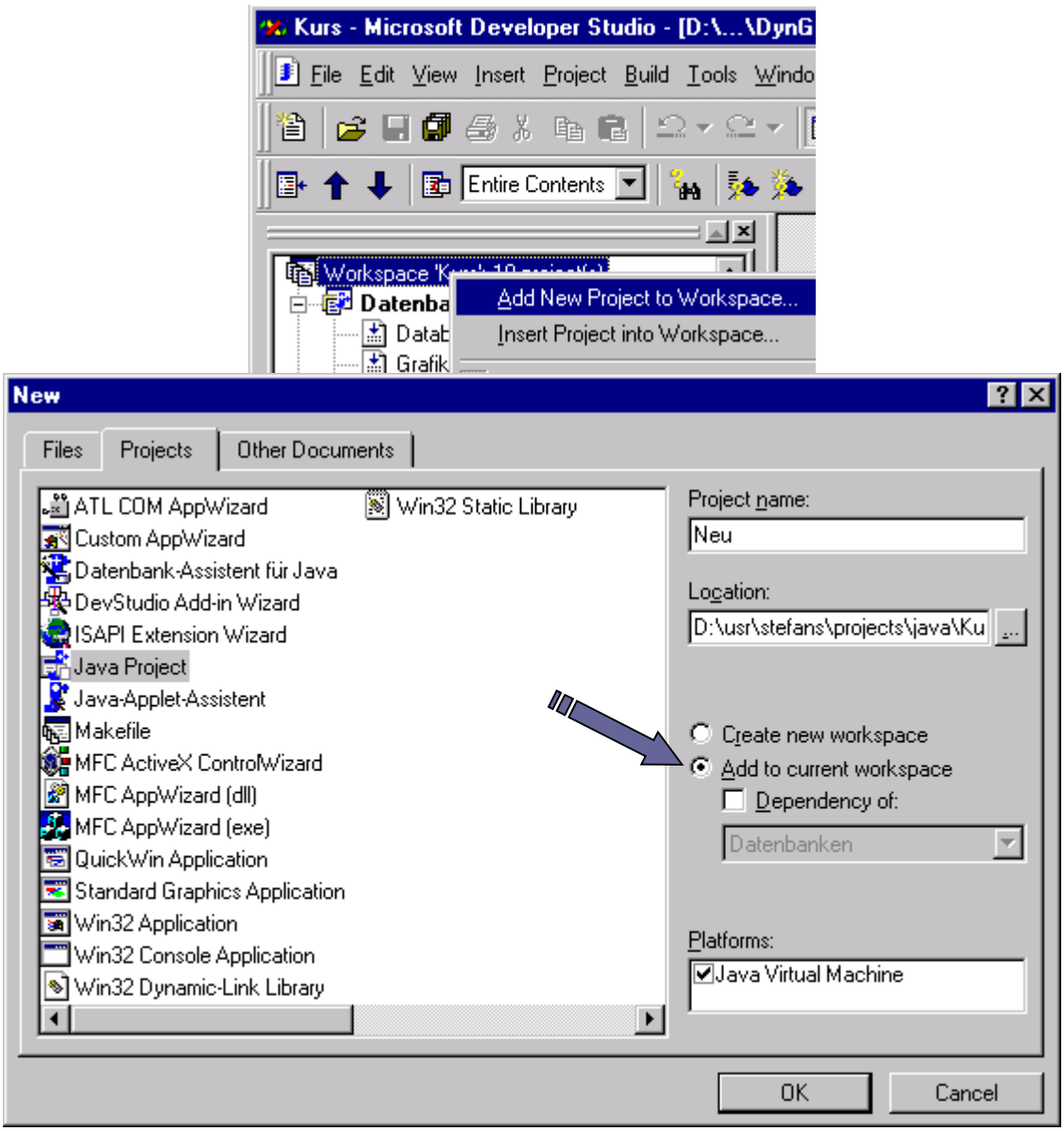


Abbildung 9-14 Neuen Workspace hinzufügen

Workspaces werden mit ihren Dateien (.java und .class) immer in eigenen Verzeichnissen abgelegt. Damit nun auch alle Projekte auf gemeinsame Java-Klassen (z.B. durch Vererbung) zugreifen können, empfiehlt sich die Verwendung eines gemeinsamen Verzeichnisses, in welchen alle kompilierten Java-Klassen (also der Bytecode) abgelegt werden (besonders wichtig bei Definition eigener Packages). Dies erfolgt innerhalb eines jeden Workspace durch:

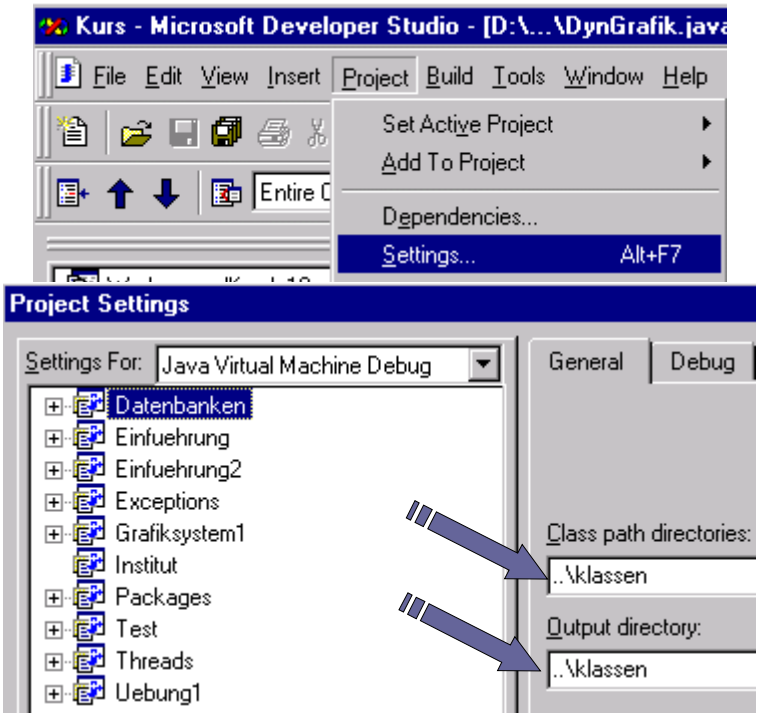


Abbildung 9-15 Gemeinsames Verzeichnis für Java-Klassen

Diese Einstellung muß für alle Workspaces (in Abbildung 9-15 im linken Fenster) separat erfolgen.

Index

Abgeleitete Klasse ... 45
Ablaufkontrolle ... 25
abstract class ... 45
Abstract Window Toolkit ... 61
Aggregation ... 106
Applets ... 9
appletviewer ... 103
Applikationen ... 9
Assoziation ... 106
Ausnahmebehandlung ... *Siehe* Exceptions
AWT ... *Siehe* Abstract Window Toolkit
Basisklasse ... 45
Bedingte Anweisung ... 26
Benutzerinteraktion ... 63
Benutzeroberflächen ... 61
  Layout ... 66
break ... 25
Browser ... 9
Bytecode ... 7
Client-Server ... 95
continue ... 25
Datenbanken ... 83
  Klassenhierarchie ... 84
do-while-Schleife ... 24
Escape-Sequenzen ... 27
Exceptions ... 36
Feinentwurf ... 53
for-Schleife ... 23
Funktionen ... 43
  Implementierung ... 47
  Statische Memberfunktionen ... 48
Grammatik ... 17
  Kommentar ... 17
  Namen ... 17
Green ... 7
if-Anweisung ... 26
Instanz ... 41
interfaces ... 47
Java
  Grammatik ... 17
  Logo ... 7
  Name ... 7
  Sicherheit ... 39
  Virtual Machine ... 7
JIT ... *Siehe* Just in time
Just in time ... 7
JVM ... *Siehe* Virtual Machine
Klasse ... 41, 105
  Abgeleitete Klasse ... 45
  Abstrakte Klasse ... 45
  Basisklasse ... 45
  Einführung ... 11
  Mehrfachvererbung ... 46
  Notation ... 49
  Schnittstellen ... 47
  Zugriff auf ... 42
Klassendesign ... 49
  Beispiel ... 52
  Feinentwurf ... 53
  Grobentwurf ... 52
  Implementierung ... 54
  Realisierung in Java ... 50
  Vorgehensweise ... 49
Klassenkonzept ... 45
Layout ... 66
  Beispiel ... 67
Layoutmanager ... 66
  BorderLayout ... 66
  CardLayout ... 66
  FlowLayout ... 66
  GridBagLayout ... 66
  GridLayout ... 66
Mehrfachvererbung ... 46
Memberfunktionen ... 41
Membervariable ... 42
multiple inheritance ... 46
Object Modeling Technique ... 105
Objekt ... 41
  Einführung ... 11
Objekthierarchie ... 62

Objektorientierte Programmierung 41
Begriffe 41
Konzepte 41
ODBC 109
OMT ..... *Siehe* Object Modeling Technique
Operatoren 19
Vorrang der 22
Packages 30
Parallele Prozesse 73
Remote Method Interface 96
Remote Method Interface Compiler 102
RMI ..... *Siehe* Remote Method Interface
Interface 97
rmic ..... *Siehe* Remote Method Interface Compiler
rmiregistry 102
Runnable 76
Schleifen 23
break 25
continue 25
do-while 24
for23
while 24
Schlüsselwörter 27
Schnittstellen 47
Sicherheit 39
SQL
DELETE 108
Insert 108
ODBC 109
Select 107
Syntax 107
Update 108
Standarddatentypen 19
Subclass 105
Superclass 105
Synchronisation 79
Thread
Interface Runnable 76
Klasse Thread 75
Synchronisation 79
Threads 73
Implementierung 73
Unterschiede zu C,C++ 28
Fehlende Features 28
Neue Features 29
Variable 17
Verteilte Anwendungen 95
Virtual Machine 7
Visual J++ 115
Workspace 115
while-Schleife 24
Workspace 115
Zuweisungsoperator 22