Softwaretechnik

Klaus Knopper

(C) 2011 Klaus.Knopper@fh-kl.de

Organisatorisches

- Vorlesung/Übung SWT Java-Teil wöchentlich jeweils Donnerstags

14:00-15:30 Uhr	Vorlesung (alle)
15:45-17:15 Uhr	Übung (Gruppe 1)
17:30-19:00 Uhr	Übung (Gruppe 2)

- Übungen mit Lösungen zu den Aufgaben der Vorwoche jeweils nach der Vorlesung.

  Mit erfolgreicher Übungsteilnahme können Bonuspunkte verdient werden, die die Note einer bestandenen Klausur aufbessern!

Zusammenfassung (1)

Nach den mathematischen und technischen Grundlagen der Informatik, und dem ersten Einblick in die Funktionsweise formaler Sprachen (Programmiersprachen) lieg der Schwerpunkt in „Softwaretechnik“ in der praktischen Programmierung und (neu) dem Verständnis der Systemtechnik bei modernen elektronischen [Datenverarbeitungs-]Geräten.

Im Programmierteil werden die Konstrukte prozeduraler und objektorientierter Programmiersprachen wiederholt und vertieft, und lösungsorientiert Ansätze implementiert.

Als „Lernsprache“ kommt im Teil 1 (Klaus Knopper) von Softwaretechnik, wie auch schon in „Grundlagen der Informatik“, größtenteils JAVA zum Einsatz, wobei jetzt vor allem die objektorientierten „Spezialitäten“ der Programmiersprache erlernt und genutzt werden.

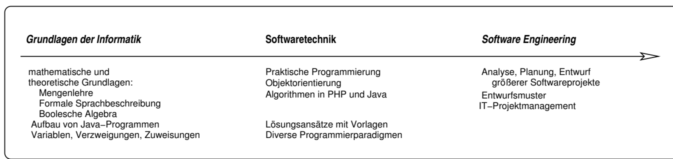
Zusammenfassung (2)

Neben der Programmierung bildet der Einblick in moderne Hardwarearchitekturen, System- und Anwendersoftware eine Rolle, um die Zusammenhänge bei Anwendungsentwicklung und Nutzung von Computersystemen besser zu verstehen, und effiziente Programme zu entwickeln.

Kursziel ist die Fähigkeit, für ein gegebenes Problem die „am besten passende“ programmiertechnische Lösung mit den erlernten Hilfsmitteln selbst erstellen zu können.

Neben PHP/Javascript/HTML im Teil 2 (Andreas Heß) für die praktische Web-Programmierung können auch weitere Programmiersprachen (z.B. Perl, Python, Bash) zum Einsatz zum Einsatz kommen, die einige Aufgaben schneller lösen können als komplexe JAVA-Programme, aber am Ende des Kurses im Gegensatz zu JAVA nicht perfekt beherrscht werden müssen.

Einordnung „Softwaretechnik“



Wiederholung: JAVA

Java ist eine **objektorientierte Programmiersprache**, die komplexe Zusammenhänge und Strukturen anhand von **Klassen** und **Objekten** modelliert, und dem Programmierer damit die Abstraktion einer Aufgabenstellung erleichtert und die Übersichtlichkeit auch großer Programme fördert. Durch Kapselung von Funktionen und Vererbung wird die Entwicklung von Programmier**schnittstellen** und **Wiederverwendbarkeit** besonders gut unterstützt, was für die Implementation von „großen“ Anwendungen (Middleware, GUIs auf dem Desktop) sehr hilfreich sein kann.

Grundsätzliches zu Java

- Der Java-Compiler (**`javac`**) erzeugt üblicherweise keinen selbstständig lauffähigen Maschinencode[a], sondern einen sogenannten *Bytecode*, zu dessen Ausführung die Java Virtuelle Maschine (JVM bzw JRE) benötigt wird.
- Java-Bytecode gibt es in einer „abgespeckten“ Form, die in einem Browser (mit Java-Support) lauffähig ist („Applet“-Klasse), sowie in der klassischen Form als Objektdatei zur Ausführung mit der JVM.

a

Vorteile von Java-Programmen

- Architektur (Rechner-) unabhängig, zur Ausführung wird lediglich eine auf der gewählten Rechnerplattform installierte und lauffähige Java Virtuelle Maschine benötigt.
- Große Menge an vordefinierten Klassen und Methoden.
- „Baukasten“-Prinzip.
- Leistungsfähige Entwicklungsumgebungen (z.B. **`eclipse`**) verfügbar.

Nachteile von Java

- Langsam, da nur interpretiert und nicht direkt als Maschinencode ausgeführt.
- Es ist eine Virtuelle Maschine notwendig, die zur Programmversion „passen“ muss.
- Inkompatibilitäten durch mangelhafte Versionierung von Klassenbibliotheken und ggf. veraltete virtuelle Maschinen.
- Wird schnell unübersichtlich ohne Entwicklungsumgebung.
- Zwar sind viele Klassenmethoden „selbsterklärend“ benannt, jedoch ist der Quelltext gegenüber beispielsweise C oft unverhältnismäßig umfangreich bei gleicher Funktionalität.

Beispiel: „Hello, World!“

```
public class hello
 {
  public static void main(String[] args)
   {
    System.out.println("Hello, World!");
   }
 }
```

Achtung: Die JVM führt standardmäßig die Klasse aus, die so heißt wie die übersetzte Datei. Der Quelltext muss hier also **`hello.java`** heißen, und der Bytecode wird nach der Übersetzung (**`javac hello.java`** erzeugt **`hello.class`**) entsprechend mit **`java hello`** ausgeführt!

Ein Applet

„Java ist einfach.“ (?)

```
import java.applet.*;
import java.awt.*;
public class Text extends Applet {
 String hello = "Hello World";
 public void paint(Graphics g) {
     g.drawString(hello, 5, 25);
  }
}
```

Dieses Applet kann, in HTML-Seiten eingebettet per **`SCRIPT`**-Tag, vom Browser ausgeführt ein Fenster mit dem Textinhalt „Hello, World!“ öffnen, eine funktionierende JVM als Browser-Plugin vorausgesetzt.

Java-Schlüsselwörter

Die folgenden Schlüsselwörter sind Bestandteil der Sprache Java, und dürfen nicht als Bezeichner für Variablen verwendet werden:

```
abstract   boolean      break   byte
case       char         class   const
continue   default      do      double
else       extends      finally float
for        goto         if      implements
import     instanceof   int     long
native     new          package private
protected  return       short   super
switch     synchronized this    throws
transient  try          void    while
```

Variablendeklarationen in Java

Im Gegensatz zu C können in Java Variablen auch zwischen Anweisungen bzw. Methodenaufrufen deklariert werden, z.B. erst unmittelbar vor ihrer ersten Benutzung.

```
public class VarDecl {
  public static void main(String[] args) {
    int a;
    a = 1;
    char b = 'x';
    System.out.println(a);
    double c = 3.1415;
    System.out.println(b); System.out.println(c);
    boolean d = false; System.out.println(d);
    for(int i = 0; i < 10; i++)
      System.out.println(i);
  }
}
```


Klassen vs. Dateien

Es ist zwar möglich, Klassen innerhalb anderer Klassen zu definieren, üblicherweise werden Klassen aber in einzelnen Dateien gespeichert, wobei der Dateiname dem Klassennamen (plus Endung **`.java`**) entspricht, unter Beachtung von Groß- und Kleinschreibung.

Wird nun in einer Klasse auf Methoden oder Variablen einer anderen Klasse zugegriffen, so versucht die Java VM die jeweilige Klasse aus einer Datei **`KlassenName.class`** nachzuladen, d.h. jede in einem Java-Programm verwendete Klasse darf (und wird üblicherweise auch) in einer separaten Datei gespeichert und übersetzt werden.

Der „Zusammenbau" des Programms und die Evaluation der Klassenmethoden und Variablen erfolgt zur Laufzeit, sofern nicht schon der Compiler die Informationen über eine Klasse benötigt, von der die korrekte Übersetzung des anderen Klasse abhängt. In diesem Fall muss der Quelltext für die jeweiligen Klassen in der richtigen Reihenfolge übersetzt werden.

`static`-Variablen und Methoden

(Nicht-statische) Variablen und Methoden in einer Klasse können erst dann benutzt werden, wenn von der Klasse ein Objekt mit **`new`** erzeugt wird.
Nur die als **`static`** deklarierten Variablen, Objekte und Methoden existieren bereits vom ersten Laden der Klasse bis zum Programmende, und können auch von mehreren Instanzen/Objekten einer Klasse gemeinsam benutzt werden, werden also nicht für jede Instanz mit **`new`** neu erzeugt. Manche von der Java VM und der Java Klassenbibliothek vorgegebene Variablen oder Methoden müssen als **`static`** definiert werden, z.B. das schon bekannte `public static void main(...)` als automatisch aufgerufene Methode.

Deklaration und Nutzung von Arrays

Arrays in Java sind Objekte, die mehrere Elemente gleichen Datentyps speicher können. Gleichzeitig enthalten Arrays als Klasse auch Methoden und Variablen, mit denen z.B. die bei der Instanzierung angegebene Größe festgestellt werden kann. Zunächst muss zuerst eine Array-Variable deklariert werden und anschließend das mit `new` erzeugte Array der Variablen zugewiesen werden.

```
int[] a;  // Deklaration Array-Variable a
// Zuweisung eines Arrays mit 5 int-Elementen
a = new int[5];
```

Die Deklaration und Initialisierung kann auch in einem Schritt durchgeführt werden.

```
int[] a = new int[5];
int[] a = { 1, 2, 3, 4, 5}; // Initialisierung
```


Zugriff auf Arrays (1)

Der Zugriff auf ein Array-Element erfolgt, wie in C, über seinen Index, beginnend mit 0.

```
public static void main(String[] args)
{
  int[] zahl = new int[2];
  zahl[0] = 2;
  zahl[1] = 3;
  System.out.println("zahl hat " + zahl.length +
                     " Elemente.");
  System.out.println(zahl[0]);
  System.out.println(zahl[1]);
  }
}
```


Zugriff auf Arrays (2)

Typisch ist der Zugriff auf Arrays mit einer `for()`-Schleife, die von 0 bis zur Arraygröße minus 1 zählt. Das folgende Beispiel gibt die Kommandozeilenparameter, mit denen das Java-Programm **`Beispiel`** aufgerufen wurde, nacheinander aus.

```
public static void main(String[] args) {
 for(int i=0; i<args.length; i++)
  System.out.println("args[" + i + "] ist: "
                    + args[i]);
}
```

Beispiel-Aufruf:

```
java Beispiel "Dies ist ein Test" und so weiter.
```

Arrays sortieren - Arrays.sort()

```
import java.util.Arrays;
...
int[] array_1 = { 10, 5, 1, 8, -1, 4 };
Arrays.sort(array_1);
```


Die **String**-Klasse (1)

In der Java-Klasse **String** sind im Gegensatz zum aus C bekannten **char *** auch Methoden definiert, die es erlauben, mit dem aus der Arithmetik bekannten Operatorzeichen **+** Zeichenketten zusammenzuhängen, oder Umwandlungen zwischen Text und Zahlen vorzunehmen.

```
public class StringVerkett
{
  public static void main(String[] args)
  {
    int a = 5;
    double x = 3.14;

    System.out.println("a = " + a);
    System.out.println("x = " + x);
  }
}
```


Die **String**-Klasse (2)

```
public class StringVergleich
 {
  public static void main(String[] args)
  {
    String a = new String("hallo");
    String b = new String("hallo");
    System.out.println("a == b liefert " + (a == b));
    System.out.println("a != b liefert " + (a != b));
  }
}
```

Die **String**-Klasse (3)

```
public class StringVergleich2
{
  public static void main(String[] args)
  {
    String a = new String("hallo");
    String b = new String("hallo");
    System.out.println("a.equals(b) liefert " +
                                a.equals(b));
  }
}
```


Von **String** nach **int**

Die Zeichenkette `"123"` ist etwas anderes als die Zahl **123**. Eine automatische Umwandlung von **String** nach **int** findet nicht statt.
In der JAVA-Bibliothek befinden sich z.B. in der Klasse **Integer** Hilfsfunktionen für die Umwandlung.

```
// Nicht erlaubt: int i = "123";
// Nicht erlaubt: String s = 123;

// OK:
String s = new Integer(123).toString();
int i = Integer.valueOf("123").intValue();
```


Integeroperationen - Beispiel

Die Standard-Klasse **Integer** in Java stellt einige praktische Methoden zur Verfügung, die mit dem Basistyp **int** nicht so einfach zu realisieren sind.

```
public class IntOut0 {
 public static void main (String args[]) {
  int k=987654321;
  System.out.println(k + " zur Basis 10 ist " +
                      Integer.toString(k));
  System.out.println(k + " zur Basis  2 ist " +
                      Integer.toBinaryString(k));
  System.out.println(k + " zur Basis  8 ist " +
                      Integer.toOctalString(k));
  System.out.println(k + " zur Basis 16 ist " +
                      Integer.toHexString(k));
  System.out.println(k + " zur Basis  3 ist " +
                      Integer.toString(k,3));
 }
}
```

Abschnitt „Betriebssysteme und Anwendungen“

1. Systemsoftware
   Hierzu gehören das Betriebssystem (inkl. „Treiber“ bzw. Kernel und -module) des Computers sowie alle Systemdienste und -programme, die dafür sorgen, dass die Hardware-Resourcen des Computers im laufenden Betrieb nutzbar und sicher sind.
2. Anwendersoftware
   Hierzu gehören die Programme, mit denen der Computer-Nutzer direkt arbeitet. Unter Unix zählt neben den Anwendungen (Office-, Datenverarbeitende Programme, Spiele, Internet-Nutzungssoftere) auch der graphische Desktop zur Anwendersoftware, und kann durch den Anwender beliebig ausgetauscht und verändert werden.

☞ Handouts „Bootvorgang“, „Linux-Distributionen“

Kompatibilität

Jedes Betriebssystem und jede Version davon stellt eine Laufzeitumgebung für Anwenderprogramme zur Verfügung. Aufgrund der unterschiedlichen Schnittstellen (APIs) sind diese leider in den meisten Fällen untereinander inkompatibel, d.h. grundsätzlich:

- Windows-Programme laufen nicht unter Linux,
- Linux-Programme laufen nicht unter Windows,
- Programme für neuere Windows-Versionen laufen nicht mit älteren Versionen zusammen,
- Programme für neuere Linux-Versionen laufen nicht mit älteren Versionen zusammen,
- Programme für eine Prozessorarchitektur laufen nicht auf einer anderen.

Hierfür gibt es einige Lösungsansätze, die das „unögliche“ möglich machen.
☞ Handout „Windows-Programme-unter-Linux-und-umgekehrt“.

Dateisystem

Um identifizierbare Objekte (Multimedia-Dateien, Dokumente, Programme, ...) auf Datenträgern schreiben und wieder von ihnen lesen zu können, werden diese bei fast allen Betriebssystemen in einem Ordnungssystem mit hierarchisch angeordneten Verzeichnissen abgelegt.

Während Windows aus historischen Gründen allen Datenträgern „Laufwerksbuchstaben“, und jedem Laufwerk eine individuelle Verzeichnisstruktur gibt, ist unter Unix nur ein einziger Verzeichnisbaum vorhanden, beginnend mit dem Wurzelverzeichnis **`/`**. Alle Datenträger, auch „Netzlaufwerke“, werden vom Administrator oder einem dafür ausgelegten Dateimanager in einen frei wählbaren Unterordner dieses Verzeichnisbaums „montiert“ (**`mount`**).

☞ Handout „Datentraegerverwaltung“

Anwenderprogramme (Beispiele) unter OSS

1. Textverarbeitung, Tabellenkalkulation, Präsentation, Zeichnungen, Datenbank-Anbindung: OpenOffice
2. Grafikbearbeitung: Gnu Image Manipulation Program (GIMP)
3. WWW: Apache WWW Server, Mozilla Firefox Browser
4. E-Mail/Groupware: Evolution
5. Multiprotokoll Chat: Pidgin
6. Videokonferenz: Ekiga

☞ Handouts „OpenOffice“, „Firefox“, „Gimp“

Distribution (1)

Eine „Distribution“ fasst bekannte Software aus dem Open Source Pool als ein in sich stimmiges „Produkt“ zusammen, das als Zusammenstellung von Betriebssystem und ausgewählter Anwendersoftware verteilt wird. Der Fokus liegt dabei auf bestimmten Zielgruppen.

Distribution (2)

Ubuntu gilt als einsteigerfreundlich und soll in der Installation und Wartung (Administration) besonders einfach zu bedienen sein.

Fedora wendet sich v.a. an Kunden aus dem unternehmerischen Bereich, die stabile Server-Software mit Zertifizierung für Standardsoftware, u.U. auch proprietäre, wünschen.

OpenSuSE war ursprünglich speziell auf Anwender im deutschsprachigen Raum spezialisiert, seit dem Kauf durch Novell jedoch auch eher am internationalen Markt ausgerichtet.

Debian ist die größte vollständig Community-basierte Distribution, setzt höheres technisches Verstöndnis voraus, und ist die Basis für viele kommerzielle Distributionen wie Ubuntu.

Knoppix ist eine auf den Betrieb als autokonfigurierendes Live-System (direkter Start von Wechselmedien oder übers Netz) spezialisierte Variante von Debian.

Rechtliche Aspekte von Software / Lizenzen

- Urheberrecht
- Überlassungsmodelle (Lizenzen)
  - Verkauf (selten)
  - Nutzung / Miete (entgeltlich oder unentgeltlich)
  - Open Source / Freie Software (weitgehende Übertragung der Verwertungsrechte auf den Lizenznehmer)
- Patente (?)

Proprietäre Software

- Der Empfänger erwirbt mit dem Kauf eine eingeschränkte, i.d.R. nicht übertragbare *Nutzungs*lizenz.
- Der Empfänger darf die Software nicht analysieren („disassemble"-Ausschlussklausel).
- Der Empfänger darf die Software nicht verändern.
- Der Empfänger darf die Software nicht weitergeben oder weiterverkaufen.

Diese Restriktionen werden im Softwarebereich so breit akzeptiert, dass man fast schon von einem „traditionellen" Modell sprechen kann.

„Freie Software"

- Freie Software stellt Software als Resource/Pool zur Verfügung.
- Freie Software sichert dem Anwender (Benutzer und Programmierer) bestimmte Freiheiten.
- Freie Software stellt eine Basis (Lizenz) für eine Zusammenarbeit von Gruppen (oder Firmen) zur Verfügung.

Was ist Freie Software/Open-Source?

- Open-Source (engl. = offene Quelle)
- Freie Software (FSF, 1984) ist Teilmenge von Open-Source-Software.
- Open-Source ist kein Produkt, sondern
- eine *Methode*, um Software zu entwicklen.
- Open-Source-Definition lt. OSI.
- „Frei“ steht für **Frei**heit (ff.), nicht für „kosten**frei**“!

Die GNU General Public License

gibt den *Empfängern* der Software das Recht, ohne Nutzungsgebühren

- die Software für alle Zwecke einzusetzen,
- die Software (mit Hilfe der Quelltexte) zu analysieren,
- die Software (mit Hilfe der Quelltexte) zu modifizieren,
- die Software in beliebiger Anzahl zu kopieren,
- die Software im Original oder in einer modifizierten Version weiterzugeben oder zu verkaufen, auch kommerziell, wobei die neuen Empfänger der Software diese ebenfalls unter den Konditionen der GPL erhalten.

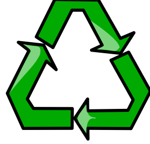
`http://www.gnu.org/`

Die GNU General Public License

- zwingt NICHT zur Veröffentlichung/Herausgabe von Programm oder Quellcode,
- zwingt NICHT zur Offenlegung ALLER Software oder Geschäftsgeheimnisse,
- verbietet NICHT die kommerzielle Nutzung oder den Verkauf der Software,
- verbietet NICHT die parallele Nutzung, oder lose Kopplung mit proprietärer Software.

Die GNU General Public License

Aber: Alle EMPFÄNGER der Software erhalten mit der GPL die gleichen Rechte an der Software, die die Mitentwickler, Distributoren und Reseller ursprünglich hatten (und weiterhin behalten).



Wer legt die Lizenz fest?

Der Urheber.

Für wen gilt eine Lizenz?

Eine Lizenz gilt für die in der Lizenz angegebenen Personenkreise (sofern nach landesspezifischen Gesetzen zulässig).
Beispiel: Die GNU GENERAL PUBLIC LICENSE gilt für

- alle legalen EMPFÄNGER der Software, die
- die Lizenz AKZEPTIERT haben.

„Wer liest schon Lizenzen?“

- Zumindest in Deutschland bedeutet das FEHLEN eines gültigen Lizenzvertrages, dass die Software NICHT ERWORBEN und NICHT EINGESETZT werden darf.
- In Deutschland gibt es seit der letzten Änderung des Urheberrechtes keine generelle Lizenz-Befreiung mehr.
- Wurde die Lizenz nicht gelesen, oder „nicht verstanden“ (weil z.B. nicht in der Landessprache des Empfängers vorhanden), so ist die rechtliche Bindung, und daraus resultierend, die Nutzungsmöglichkeit der Software, formal nicht gegeben.

Auch als „Freeware“ deklarierte Software ist hier keine Ausnahme. Wenn keine Lizenz beiliegt, die eine bestimmte Nutzungsart ausdrücklich ERLAUBT, gilt sie als VERBOTEN.

„Kopierschutz“ (1)

- Soll die nicht vom Rechteverwerter genehmigte Vervielfältigung unterbinden,
- ist de facto technisch überhaupt nicht realisierbar,*)
- ein „wirksamer“ Kopierschutz (juristisch genügt die Angabe auf der Packung, technisch kann der Kopierschutz absolut wirkungslos sein) darf nach der Urheberrechtsnovelle von 2003 nicht mehr umgangen werden, auch nicht zum Anfertigen einer Kopie für den Privatgebrauch,

...

*) Alles, was audiovisuell wahrgenommen werden kann, kann auch kopiert werden, notfalls über die „analoge Lücke“.

„Kopierschutz“ (2)

- dennoch bleibt das Umgehen eines Kopierschutzes zur ausschließlichen Eigennutzung nach §108b UrhG aber straffrei,
- und laut §69a Abs. 5 UrhG ist das Umgehen einer Kopiersperre speziell bei Computerprogrammen auch nicht in jedem Falle ein Strafdelikt (VORSICHT!).

Fragwürdige Lizenzklauseln

- Genereller „Haftungsausschluss“,
- Eigentumsvorbehalt,
- Konkurrenzausschluss,
- Abtreten von gesetzlich garantierten Grundrechten.

Autor/Distributor haften...

- für „Geschenke“ nur bei GROBER FAHRLÄSSIGKEIT,
- für „Verkäufe“ bei allen vom Verkäufer/Hersteller verschuldeten Fehlern.

GPL-Verträglichkeit

- GPL erlaubt die Integration proprietärer Software auf dem gleichen Datenträger, solange die nicht-GPL-Komponenten wieder separierbar sind (Beispiel: KNOPPIX-CD, versch. Linux-Distributionen).
- BSD-Lizenz erlaubt die Integration von Code in proprietäre Programme ohne Offenlegungspflicht. Es muss lediglich darauf hingewiesen werden, dass die Software BSD-Komponenten enthält (Beispiel: TCP/IP-Stack im Windows-Betriebssystem).
- Die Programm-Urheber können für ihr Werk auch eine Auswahl verschiedener Lizenzen „zum Aussuchen“ anbieten (Dual Licensing).

Tabelle: Lizenzmodelle und Rechte

	Nutzung kostenlos	frei kopierbar	zeitlich unbegrenzt nutzbar	Quelltext wird mitgeliefert	Modifikation erlaubt	Einbau in prop. Produkte erlaubt	Derivate mit anderen Lizenzen mögl.
proprietäre Software							
Shareware	✔	✔					
Freeware	✔	✔	✔				
GPL	✔	✔	✔	✔	✔		
LGPL	✔	✔	✔	✔	✔	✔	
BSD	✔	✔	✔	✔	✔	✔	✔

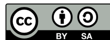
Geld verdienen mit Open Source

Da das Einkassieren von „Nutzungslizenzgebühren“ unter Open Source nicht zulässig ist, und die Verbreitung (Kopie, Weiterbearbeitung etc.) auch nicht eingeschränkt werden kann, ist das Geschäftsmodell bei Open Source:

- ➭ Nicht die Software selbst, sondern eine Dienstleistung als Produkt anbieten (Support, Wartung, Anpassung),
- ➭ nicht „Software von der Stange“ verkaufen, sondern Software im Auftrag entwickeln bzw. auf Kundenbedürfnisse individuell anpassen (Baukasten-Prinzip).

☞ Der Großteil des Umsatzes der bekannten „Software-Riesen“ baut auf diesem Konzept auf, wobei der Anteil an eingesetzter Open Source Software aber unterschiedlich hoch ist.

Creative Commons



- ➭ Verschärfung des Urheberrechtes zugunsten der Rechteverwerter-Industrie führt zu Ablehnung durch viele Kreative.
- ➭ Schaffung von rechtlichen Grundlagen zur Eigenvermarktung und Eigenverlag von Kunstwerken durch die Künstler ohne Exklusivvertrag mit einer Verwertungsgesellschaft.
- ➭ „Lizenz-Baukasten“ für verschiedene Empfängerkreise und Verwertungszwecke.

Beispiel für prof. Animationsfilme unter Creative Commons Lizenz: „Elephants Dream“, Big Buck Bunny, Sintel (neu).

Weitere Literatur zum Internetrecht

Professor Dr. Thomas Hoeren, Institut für Informations-, Telekommunikations- und Medienrecht an der Universität Münster, Kompendium zum Internetrecht (PDF)

...zurück zu Java.

Weitere, juristische und andere nicht-technische Aspekte der Softwaretechnik folgen in der Vorlesung „Software-Engineering.“.

Paradigmen

Ein Paradigma bezeichnet einen *Kernsatz*, der einen Sachverhalt beschreibt oder modelliert. Üblicherweise wird in einem Paradigma in einem Satz zusammengefasst, worum es geht.

In der Programmierung bezeichnet ein Paradigma die Herangehensweise an ein Problem. Für unterschiedliche Probleme sind unterschiedliche Lösungsansätze optimal, und dementsprechend gibt es für jede Aufgabe auch eine „am besten passende“ Programmiersprache.

Imperatives Programmierparadigma

„Befehle werden ausgeführt.“

Beispiel in der Shell (bash):

```
read -p "Bitte x eingeben: " x
let xx=x*x
echo "x zum Quadrat ist" $xx
```


Prozedurales Programmierparadigma

„Oft verwendete Befehlsfolgen werden zu Prozeduren und Funktionen zusammengefasst.“

Beispiel in C:

```
int quadrat(int x) { return x*x; }
...
int c2 = quadrat(a) + quadrat(b);
```


Funktionales Programmierparadigma

„Die Lösung steckt in der Definition.“

Beispiel in Haskell:

```
fib :: Integer -> Integer
fib(0) = 0
fib(1) = 1
fib(n+2) = fib(n) + fib(n+1)
Main> fib(30)
```

Objektorientiertes Programmierparadigma

„Objekte beschreiben konkrete Dinge durch Eigenschaften (Attribute) und Methoden.“

Beispiel in JAVA:

```
public class Auto extends Fahrzeug {
  Motor m;
  Räder[4] r;
  ...
  public boolean start_motor();
  public void tanken(Benzinart b);
  ...
}
```


Sichtbarkeit von Objektattributen in Java

Mit dem Schlüsselwort **`private`** vor einer Variablen innerhalb einer Klasse wird dafür gesorgt, dass diese Variable nur für die Klasse selbst und deren Klassenmethoden sichtbar ist. **`public`** sorgt hingegen dafür, dass auch Objekte von anderen Klassen Zugriff auf die Variablen bekommen.
Es gilt als guter Stil, klasseninterne Variablen stets als **`private`** zu deklarieren (Voreinstellung), und **`public`**-Methoden zu definieren, die einen kontrollierten Zugriff auf die privaten Variablen von außen erlauben.

```
public class Pizza {
 private int scharf;
 public int getPfefferMenge() { return scharf; }
 public void setPfefferMenge(int wieviel) {
  if(wieviel > 10) wieviel=10; scharf = wieviel;
 }
}
```


Objekterzeugung in Java

In Java sind Objektvariablen zunächst *Referenzen* (ähnlich wie Zeiger in C), denen entweder ein bereits existierendes Objekt (bzw. dessen Referenz) zugewiesen werden kann, oder ein *neues* Objekt, was mit dem Java-Schlüsselwort **`new`** und dem Aufruf des *Konstruktors* erzeugt wird.

```
public class Test {

 public class NeuesObjekt {
   public int zahl;
 }

 public static void main(String[] args) {
   NeuesObjekt n = new NeuesObjekt();
   n.zahl = 1;
 }
}
```

Konstruktor

Jede Klasse enthält, auch wenn es nicht explizit angegeben ist, (mindestens) eine Methode, die so heißt wie die Klasse selbst, und die beim Erzeugen eines Objektes dieser Klasse automatisch aufgerufen wird. Man kann diese Methode selbst (über-)schreiben, und mit Parametern versehen:

```
public class Auto
{
  public String name;
  public int    erstzulassung;
  public int    leistung;
  public Auto(String name) { this.name = name; }
}
```

Später kann dann ein Objekt der Klasse **`Auto`** mit `Auto a = new Auto("Flitzer");` erzeugt werden, wodurch die Variable **`name`** mit dem String "`Flitzer`" vorbelegt wird. **`this`** ist stets eine Referenz auf das aktuelle Objekt, und kann normalerweise entfallen (allerdings nicht in diesem Beispiel, warum?).

Destruktor

Jede Klasse enthält, auch wenn es nicht explizit angegeben ist, eine Methode, die nach Beendigung der Lebenszeit eines Objektes dieser Klasse automatisch von der sog. „Garbage Collection“ der Java VM aufgerufen wird. Der genaue Zeitpunkt ist allerdings nicht vorhersehbar.

```
public class Test {
  public class KonstruktorTest {
    KonstruktorTest()  {
      System.out.println("Konstruktor wurde aufgerufen!");
    }
  protected void finalize() {
      System.out.println("Destruktor wurde aufgerufen!");
    }
  }
  public static void main(String[] args) {
    KonstruktorTest t1 = new KonstruktorTest();
  }
}
```


Grafik unter Java - die `Graphics()`-Klasse

Viele grundlegende Grafikfunktionen von Java sind in der **`Graphics()`**-Klasse untergebracht, die zum Paket **`java.awt`** gehört.
Die **`Graphics()`**-Klasse umfasst u.a. Methoden zum Zeichnen von Linien, Kreisen, Ellipsen, Rechtecken und Texten in verschiedenen Farben und Schriften. Die meisten dieser Funktionen zeichnen in ein (x,y)-Koordinatensystem und benötigen ein Graphics-Objekt als Referenz für die Zeichenfläche (Fenster, rechteckiger Bereich im Browser o.ä.).

Graphics()-Objekt

Aus grafikfähigen Objekten kann das zugehörige **Graphics**-Objekt mit der Funktion **getGraphics()** ermittelt werden.

```
import java.awt.*;
...
Frame f = new Frame(); // Fenster-Objekt
f.setSize(400, 400); // Größe festlegen
f.setVisible(true); // jetzt öffnen!
Graphics g = f.getGraphics();
```

Bei Applets wird in der Methode **paint()** automatisch ein **Graphics**-Objekt als einziges Argument übergeben.

Applets

Die Applet-Klasse gehört zur Standard Java-API und ist mit dem Java-Plugin für Webbrowser verbunden. Der Browser stellt dem Applet einen vordefinierten Bereich zum Zeichnen zur Verfügung, innerhalb desse es sich „austoben“ kann. Ein Applet kann jedoch auch neue Fenster öffnen.
Im Gegensatz zu einer Standalone-Anwendung wird bei Applets bereits vom Java-Plugin ein Objekt erzeugt und bestimmte Funktionen werden innerhalb des Applets bei vordefinierten Zustandsänderungen automatisch aufgerufen.

Grundgerüst eines Applets

```
// Importieren der notwendigen awt-Klassen
public [Klassenname] extends java.applet.Applet {
// Variablen-Deklaration und Definition von Methoden
 public void init() { ... }  // beim Laden des Applets
 public void start() { ... } // beim Start des Applets
 public void stop() { ... }  // wenn HTML-Seite mit Applet
                             // verlassen wird
 public void destroy() { ... } // Löschen des Applets und
                               // Freigeben der Resourcen
 public void paint(Graphics g) { ... } // wird aufgerufen,
                       // wenn Applet (neu) gezeichnet wird
 public void run() { ... }   // bei Multithreading anstelle
                             // von paint()
 ...
```

Graphics-Funktionen (1)

... stehen in jedem **Graphics**-Objekt zur Verfügung:

drawString("Zeichenkette", x, y)
Setzt "Zeichenkette" an Position **x,y**

drawLine(x_1, y_1, x_2, y_2 **)**
Linie von Punkt x_1, y_1 zu Punkt x_2, y_2 zeichnen

drawRect(x,y,b,h)
Rechteck von Punkt **x,y** beginnend mit Breite **b** und Höhe **h** zeichnen

fillRect(x,y,b,h)
wie **drawRect()**, aber gefülltes Rechteck

draw3DRect(x,y,b,h,true)
wie **drawRect()**, aber mit "Relief"-artiger Farbgebung, wenn letzter Parameter WAHR

Graphics-Funktionen (2)

drawRoundRect(x,y,b,h, r_b**,** r_h**)**
wie **drawRect()**, mit abgerundeten Ecken. Die beiden letzten Parameter bezeichnen die Breite und Höhe der Kreisbögen

drawPolygon(poly)
Polygonzug. Der Parameter **Polygon poly** enthält die Koordinaten der Eckpunkte.

```
// Definiere ein Array mit X-Koordinaten
int[] xCoords = { 10, 40, 60, 300, 10, 30, 88 };
// Definiere ein Array mit Y-Koordinaten
int[] yCoords = { 20, 0, 10, 60, 40, 121, 42 };
// Bestimme Anzahl Ecken über Methode length
int anzahlEcken = xCoords.length;
Polygon poly = new Polygon(xCoords, yCoords, anzahlEcken);
drawPolygon(poly); // Zeichne das 7-Eck
```

fillPolygon(poly)
wie **drawPolygon(poly)**, aber gefüllt.

Graphics-Funktionen (3)

drawOval(x, y, r_x**,** r_y**)**
Ellipse oder Kreis um Punkt **x,y** Zeichnen unter Angabe der zwei Radien.

fillOval(x, y, r_x**,** r_y**)**
wie **drawOval()**, aber gefüllt.

Farbe setzen mit **`setColor()`**

```
import java.awt.*;
...
 // Erzeugen eines Farbobjektes
 Color pinkColor = new Color(255, 192, 192);

 // Setze Farbe und zeichne einen gefüllten Kreis
 setColor(pinkColor); fillOval(250, 150, 150, 150);

 // Verwendung einer Farbkonstanten
 setColor(Color.green); drawRect(40,40,100,200);
```


Erstellen von Fontobjekten, Beispiel

```
import java.awt.Font;
import java.awt.Graphics;
public class FontTest extends java.applet.Applet {
 public void paint(Graphics g) {
  Font f = new Font("TimesRoman", Font.PLAIN, 18);
  Font fb = new Font("TimesRoman", Font.BOLD, 18);
  Font f2i = new Font("Arial", Font.ITALIC, 34);
  g.setFont(f);
  g.drawString("Normal (plain) TimesRoman", 10, 25);
  g.setFont(fb);
  g.drawString("Fett (bold) TimesRoman", 10, 50);
 }
}
```


Bilder laden und anzeigen

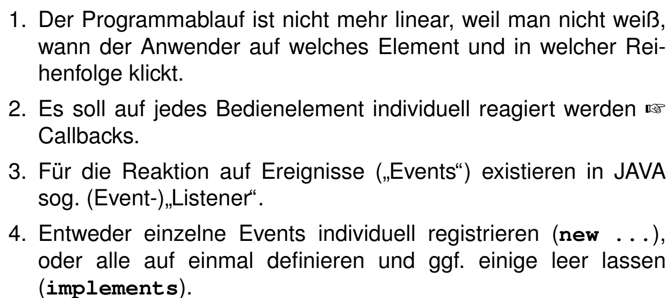
```
import java.awt.Image;
import java.awt.Graphics;
public class DrawImage2 extends java.applet.Applet {
 Image bild;
 public void init() {
 // Bild laden - ein jpg-File
  bild = getImage(getDocumentBase(), "bild.jpg");
 }
 public void paint(Graphics g) {
  g.drawImage(bild, 0, 0, this);
 }
}
```

Animationen unter Java

```
import java.awt.Image;
import java.awt.Graphics;
public class Animation1 extends java.applet.Applet {
 Image bild;
 public void init() {
  bild = getImage(getCodeBase(), "bild.gif");
  resize(600, 200); }
 public void paint(Graphics g) {
  for (int i=0; i < 1000; i++) {
    int bildbreite = bild.getWidth(this);
    int bildhoehe = bild.getHeight(this);
    int xpos = 10;  // Startposition X
    int ypos = 10;  // Startposition Y
    g.drawImage(bild, (int)(xpos + (i/2)),
     (int)(ypos + (i/10)), (int)(bildbreite * (1 +
     (i/1000))),(int) (bildhoehe * (1 + (1/1000))),
     this); } } }
```


GUI-Programmierung

Bei der Einführung von interaktiven Elementen muss einiges beachtet werden:



1. Der Programmablauf ist nicht mehr linear, weil man nicht weiß, wann der Anwender auf welches Element und in welcher Reihenfolge klickt.
2. Es soll auf jedes Bedienelement individuell reagiert werden ☞ Callbacks.
3. Für die Reaktion auf Ereignisse („Events“) existieren in JAVA sog. (Event-)„Listener“.
4. Entweder einzelne Events individuell registrieren (**`new ...`**), oder alle auf einmal definieren und ggf. einige leer lassen (**`implements`**).

GUI-Programmierung: Bedienelemente

Die Bedienelemente werden mit **`add(element)`** mit dem jeweiligen Fenster (**`Frame`**) verbunden und benachrichtigen den zugehörigen Event-Handler.

Klasse	**Erklärung**	**Event-Handler**
`Button`	Schaltfläche (Klick)	**`ActionListener`**
`TextField`	Einzeiliges Texteingabefeld	**`ActionListener`**
`TextArea`	Mehrzeiliges Texteingabefeld	**`TextListener`**
`Choice`	Aufklapp-Menü	**`ItemListener`**
`Checkbox[Group]`	Auswahl[Radio]-Kästchen	**`ItemListener`**
`MenuBar`	Fenster-Menü	**`ItemListener`**
`Frame`	Fenster allgemein	**`WindowListener`**, **`MouseListener`**, **`MouseMotionListener`**

GUI-Programmierung - Interfaces (1)

In den Event-Interfaces sind die Namen der Funktionen, die bei event-gesteuerten Bedienelementen auftreten können, definiert. Implementiert eine Klasse ein Interface, müssen ALLE Methoden aus dem Interface selbst programmiert werden, dürfen aber auch leer sein.

```
import java.awt.*; import java.awt.event.*;
public class Fenster implements WindowListener {
 public Fenster(){ // Konstruktor
  Frame f = new Frame("Neues Fenster");
  f.setSize(200,100);
  f.setVisible(true);
  f.addWindowListener(this); // Siehe (2)
 }
 public static void main(String[] args) {
  new Fenster();
 }
 ...
```


GUI-Programmierung - Interfaces (2)

```
 public void windowOpened(WindowEvent e ) {}
 public void windowActivated(WindowEvent e) {}
 public void windowDeactivated(WindowEvent e) {}
 public void windowIconified(WindowEvent e) {}
 public void windowDeiconified(WindowEvent e) {}
 public void windowClosed(WindowEvent e) {}
 public void windowClosing(WindowEvent e) {
  e.getWindow().dispose(); // Fenster schließen
  System.exit(0);          // Programm beenden
 }
} // Ende Klasse Fenster
```


GUI-Programmierung - Adapter-Klassen (1)

Adapter-Klassen in Java AWT enthalten leere „Default“-Implementierungen der zuvor behandelten MouseListener, WindowsListener, ... Interfaces und können so mit **`addMouseListener()`** usw. den interaktiven Elementen hinzugefügt werden, ohne dass jede Funktion selbst implementiert werden muss. Die leeren Methoden der Adapter-Klassen können einzeln durch eigene überschrieben werden, schon beim Erzeugen eines Objektes mit dem Konstruktor:

```
MouseAdapter m = new MouseAdapter() {
 public void mouseClicked(MouseEvent e) {
  System.out.println("Knopf wurde gedrückt.");
  System.exit(0);
 }

Button b = new Button("Klick hier!");
b.addMouseListener(m);
}
```

GUI-Programmierung - Adapter-Klassen (2)

Ein vollständiges Programm:

```
import java.awt.*; import java.awt.event.*;
public class HelloButton {
 public static void main(String[] args) {
  Frame frame = new Frame("Mein Frame"); // Fenster
  frame.setSize(400,200); // Größe ändern (Breite, Höhe)
  Button button = new Button("Klick hier!");
  button.addMouseListener( new MouseAdapter() {
   public void mouseClicked(MouseEvent e) {
    System.out.println("Knopf wurde gedrückt.");
    System.exit(0);
   } } );
  frame.add(button);
  frame.setVisible(true);
 }
}
```


Textein- und ausgabe - Fenster

Texteingabe als grafische Applikation:

```
import java.awt.*; import java.awt.event.*;
public class TextEinAusgabe {
 static final Frame fenster = new Frame("Texteingabe");
 static final TextField eingabefeld = new TextField();
 static final ActionListener aktion = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
   fenster.remove(eingabefeld);
   fenster.add(new Label("Sie haben >"
             + eingabefeld.getText() + "< eingegeben"));
   fenster.setVisible(true); // Refresh erzwingen
  } };
 public static void main(String[] args) {
  fenster.add(eingabefeld);
  eingabefeld.addActionListener(aktion);
  fenster.setSize(400, 80); fenster.setVisible(true);
 }
}
```


Textein- und ausgabe - Applet (1)

Text- und Ausgabe als Applet, eingebettet in eine HTML-Seite:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class TextEinAusgabe extends Applet
{
  TextField eingabe;
  Label     ausgabe;
  public void init() {
    Label hinweis = new Label( "Oben Text eingeben" );
    eingabe = new TextField( "    " );
    ausgabe = new Label();
    setLayout( new BorderLayout() );
    add( BorderLayout.NORTH,  eingabe );
    add( BorderLayout.CENTER, hinweis );
    add( BorderLayout.SOUTH,  ausgabe );
```

Textein- und ausgabe - Applet (2)

```
    eingabe.addActionListener(
      new ActionListener() {
        public void actionPerformed( ActionEvent ev ) {
          meineMethode(); } } );
  }
  void meineMethode() {
    ausgabe.setText( "Der Text lautet: " +
                              eingabe.getText() );
  }
}
```


Fehler und Ausnahmebehandlungen

In Java lassen sich „ungewöhnliche" Ereignisse wie die vorzeitige Beendigung von Tastatureingaben, Fehler beim Lesen und Schreiben von Dateien, Division durch 0 etc. mit Hilfe sogenannter „Exceptions" abbilden und unter Verwendung von **`try`** ... **`catch`** Ausnahmebehandlungen definieren.

```
try {
  Anweisungsblock, in dem Ausnahmen auftreten können
    }
catch(Ausnahmenart 1) { Ausnahmebehandlung 1 }
catch(Ausnahmenart 2) { Ausnahmebehandlung 2 }
catch(Ausnahmenart 3) { Ausnahmebehandlung 2 }
```

Die genaue Fehlerart kann als Parameter in **`catch()`** explizit angegeben werden (z.B. **`ArrayOutOfBoundsException e`**). Die Basisklasse **`Exception`** kann hierbei für „catch-all" verwendet werden.

Fehler und Ausnahmebehandlungen

Man kann auch durch gezieltes „Werfen" von Fehlersignalen die Standardfehlerbehandlung der Java VM auslösen, oder durch spätere **`try ... catch`** Blöcke behandeln.

```
public class Ausnahmen
{
  public static void main(String[] args)
  {
    if (args.length < 2) {
      throw new IllegalArgumentException();
    }
    ...
```

Dateien schreiben in Java

```
import java.io.*;
public class DateiSchreiben {

  public static void main(String[] args) {

    try {
      FileWriter fw = new FileWriter("ausgabe.dat");
      fw.write("Hallo, Welt!");
      fw.close();
    } catch(Exception e) {
      System.err.println("Fehler: " + e.toString() );
      System.err.println("Details: ");
      e.printStackTrace();
    }

  }
}
```


Dateien lesen in Java

```
import java.io.*;
public class DateiLesen {
  public static void main(String[] args) {
    try {
      FileReader fr = new FileReader("eingabe.dat");
      int zeichen = -1;
      while(true) {
       zeichen = fr.read();
       if(zeichen < 0) break;
       System.out.print((char)zeichen);
      }
      fr.close();
    } catch(Exception e) {
      System.err.println("Fehler: " + e.toString() );
    }
  }
}
```


Ausnahmebehandlungen bei der Eingabe

Das Einlesen von Variablen ist in Java, verglichen mit anderen Programmiersprachen, verhältnismäßig kompliziert, da für die verschiedenen Eingabemethoden zunächst Eingabe„objekte“ erzeugt werden müssen und Fehler bei der Dateioperationen mit **`try`**...**`catch`**-Konstruktionen abgefangen werden müssen.

Es bietet sich an, für häufig verwendete Einleseoperationen eigene Klassen und entsprechende Klassenmethoden anzulegen, die später von anderen Klassen einfach aufgerufen werden können.
☞ **`Eingabe.java`**

Textein- und ausgabe

Textein- und ausgabe über die Kommandozeile:

```
import java.io.*; // io-Klassen laden
public class TextEinAusgabe {
 public static void main(String[] args) {
  System.out.println("Text eingeben:");
  try {
    BufferedReader in = new BufferedReader(
             new InputStreamReader(System.in) );
    String s = in.readLine();
    System.out.println("Der Text lautet: " + s);
  } catch( IOException ex ) {
    System.out.println( ex.getMessage() );
  }
 }
}
```


Arrays vs. Listen

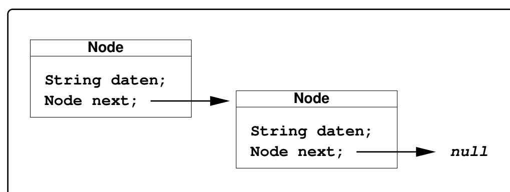
Nachteile von Arrays:

- Konstante Anzahl Elemente nach Initialisierung,
- „Einfügen“ von Elementen an bestimmter Stelle schwierig, da nachfolgende Elemente nicht ohne weiteres „verschoben“ werden können,
- „Löschen“ von Elementen schwierig, da nachfolgende Elemente nicht „aufrücken“ können.

Lösung: „Verketten“ von Objeken

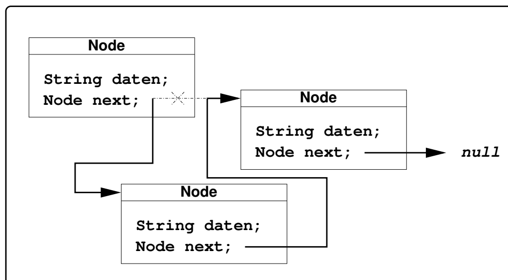
```
public class Node {
 String daten;
 Node next; // Zeiger auf NÄCHSTES Element
}

Node n = new Node();
n.next = new Node();
n.next.next = null;
```



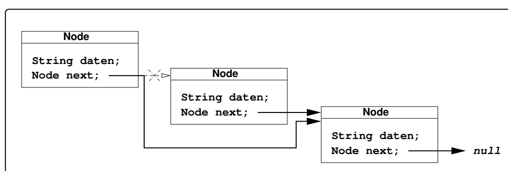
Hinzufügen eines Elements

```
Node neu = new Node();
neu.next = vorgaenger.next;
vorgaenger.next = neu;
```



Entfernen eines Elements

```
vorgaenger.next = vorgaenger.next.next;
```



Suchen eines Elements

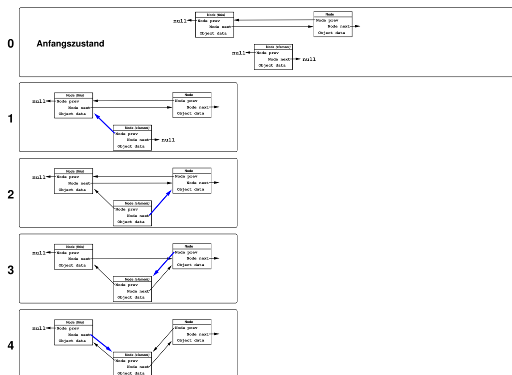
Da es nicht wie bei einem Array einen „Laufindex" **`array[i]`** gibt, muss den **`next`**-Zeigern bis zum Ende der Liste gefolgt werden.

Beispiel: Stringsuche

```
Node n = listenanfang;
while(n != null) {
 if(n.data.equals("Suchbegriff")) break; // gefunden!
 n = n.next;
}
```

`n` ist nach der Schleife eine Referenz auf das gesuchte Element, oder **`null`**.

Doppelt verkettete Liste, Einfügen



Vererbung von Klasseneigenschaften

Ein Kaffeemaschine hat eine Funktion zur Befüllung von Tassen. Eine spezielle Kaffeemaschine Cafe2000 hat zusätzlich einen Timer.

```
class Kaffeemaschine
{
  int tasse;
  public void befüllung(int wieviel) {
    tasse = wieviel;
  }
}
class Cafe2000 extends Kaffeemaschine
{
  Time t;
  pubic void timer(Time time) {
    t = time;
  }
}
```


Zuweisungsverträglichkeit von Objekten

Ein von einer abgeleiteten Klasse stammendes Objekt kann einer Variable der Basisklasse zugewiesen werden.

```
Kaffeemaschine k = new Cafe2000();
```

Um auf die erweiterten Eigenschaften zuzugreifen, ist ein cast notwendig:

```
 ((Cafe2000) k).timer(10);
```

Die Zuweisung eines Objektes der Basisklasse an eine Variable der abgeleiteten Klasse ist in Java NICHT zulässig:

falsch!

```
Cafe2000 c = new Kaffeemaschine();
```

Polymorphie

```
class PTier {
  public String  farbe;
  String kennung() {
    return "Keine Ahnung was ich mal werde.";
  }
}
class PPinguin extends PTier {
  String kennung() { return "Ich bin ein Pinguin."; }
}
...
PPinguin pingu = new PPinguin();
System.out.println( pingu.kennung() );
```

Frage: Was würde ausgegeben werden für die Fälle
`PTier pingu = new PTier();` oder
`PTier pingu = new PPinguin();`?

`abstract` Klassen und Methoden

Mit dem Java-Attribut **`abstract`** werden solche Klassen oder Methoden gekennzeichnet, die nur an abgeleitete Klassen vererbt, aber nicht direkt in Objekten implementiert werden können.

```
public abstract class Pizza {
 ...
}

// Pizza p = new Pizza(); ist NICHT ERLAUBT!!!

public class Speziale extends Pizza { // OK
 ...
}
```


Warum **`abstract`**?

- Vermeidet „unpräzise“ Objekte, die direkt von einer nicht-konkreten (eben „abstrakten“) Klasse abgeleitet werden.
- Zwingt bzw. erinnert den Programmierer, dass er die entsprechenden Klasseneigenschaften bzw. Methoden (erst) in den abgeleiteten Klassen/Klassenmethoden im Detail implementieren muss.
- Hilft, „ähnliche“ Objekte zu definieren, die aber nicht zueinander kompatibel sind.

interface-Klassen

„Eine abstrakte Klasse, die nur abstrakte Methoden/Funktionen enthält.“

```
public interface music_handler {
 Musikstück[] liste();
 Musikstück runterladen(URL u, String name);
 Musikstück runterladen(Gerät g, String name);
 void abspielen(Musikstück m);
 void übertragen(Musikstück m, Gerät g);
 void löschen(Musikstück m);
}

public class MusicPlayer
  implements music_handler {
...
}
```


Warum **interface**-Klassen?

- Ähnlich **abstract**, erfordern eine konkrete Implementierung in den abgeleiteten Klassen.
- Bieten einen Ausweg aus der fehlenden „Mehrfachvererbung“, d.h. der Tatsache, dass in Java eine Unterklasse nur von einer Basisklasse erben kann. Mehrfache, durch Komma getrennte Interfaces hinter **implements** sind aber möglich.
- Spezifizieren eher das **Verhalten** (Methoden + Aktionen) von Klassen als gemeinsame **Eigenschaften**.

Beispiele aus der Java-API

```
public class SystemProperties
  {
    public static void main( String[] args )
    {
      System.out.println(
       System.getProperties().toString()
           .replace( ',', '\n' ).replace( '{', ' '  )
           .replace( '}', ' '  ) );
    }
  }
```

NB: Jedes **replace()** liefert hier einen **String** zurück, in dem wieder die Methode **replace()** aufgerufen werden kann.

„Syntax“ (Grundlagen Wdh.)

Die Syntax einer Programmiersprache kann mit Hilfe von Regeln beschrieben werden. Diese Regeln können sowohl

- umgangssprachlich oder
- mit Hilfe einer formalen Methode

beschrieben werden. Bei mathematischen Definitionen und anderen exakten Festlegungen wird eher die formale Methode bevorzugt.

Syntax

Formale Methoden, um eine Syntax zu beschreiben, sind:

- Syntaxdiagramme,
- Formale Sprachen (z.B. Chomsky Typ 0-3 Grammatik).

Wie ist beispielsweise die Syntax aller natürlichen Zahlen definiert?

Grammatiken und Programmiersprachen

- Jede Programmiersprache besitzt eine Grammatik.
- Mit der Grammatik kann die Syntax von Programmen geprüft werden (Bestandteil des Compilers).
- Mit Hilfe der Grammatik kann eine Programmiersprache „syntaktisch“ verstanden werden.

Semantik

Die Semantik beschreibt schlicht und einfach die *Bedeutung* einer Darstellung.
Beispiele:

Formale Darstellung	Semantik
A == B	Es wird geprüft, ob **A** und **B** gleich sind.
C = A - B	Der Variablen **C** wird das Ergebnis der Subtraktion **A - B** zugewiesen.
A a = new A()	Der Referenz **a** wird ein neues Objekt vom Typ der Klasse **A** zugewiesen.

Syntax und Semantik

Während *syntaktische* Fehler vom Compiler aufgrund algorithmischer Regeln präzise gefunden werden können, sind *semantische* Fehler erst aus dem Zusammenhang zwischen erwartetem und tatsächlichem Ergebnis zu erkennen, vor allem deswegen, weil sie oft auf Missverständnissen oder falscher Interpretation von vorgegebenen Algorithmen und Anwendungsbeschreibungen durch den Programmierer beruhen.

D.h.: Nicht jedes *syntaktisch* einwandfreie (also vom Compiler übersetzbare) Programm ist automatisch fehlerfrei!

Rekursion

Rekursiv heißt eine *Funktion, die sich selbst aufruft*.

Beispiel: Die Fakultäts-Funktion

Math.:

$$\begin{aligned} \mathsf{F}(x) &= x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot 1 \\ &= \begin{cases} 1 & \text{falls} \quad x \le 1 \quad \text{(Rekursionsanfang)} \\ x \cdot \mathsf{F}(x-1) & \text{sonst} \quad \text{(Rekursionsschritt)} \end{cases} \end{aligned}$$

JAVA:

```
public int f(int x) {
 if(x <= 1) { return 1; } // Abbruchbedingung
 else       { return x * f(x-1); }
}
```

Rekursion

Übung: Aufaddieren der Zahlen von 1 ... x ohne **`for()`**-Schleife.

```
public int reihe(int x) {
 if(             ) return        ;
 return x + reihe(              );
}
```