Programmieren (Java)

Grundzüge und Leitlinien

K.O. Linn

Lernziele

- Good Programming Practice
- Datentypen: int, double, char, boolean, big decimal, Pointer, Objekte
- Operatoren: + - * / % | & || && > >> >>>
- Sprachkonstrukte: if-else, for, try-catch, Sprünge, Labels
- Sichtbarkeit von Variablen
- Methoden und Funktionen
- Unterschied zw. Funktionen- und Objekt- orientierter Denkweise.
- Erstellen eines einfachen Programms
- Umsetzen eines Algorithmus in einen Programmablauf
- Applets

Lernziele(Forts.)

- Wissen was ist ein(e):
  - Pointer/Referenz
  - Garbage Collector
  - Stack
  - Hash table
  - Vector
  - String, Character/Byte-Array
  - Hex-, Dez-, Bin- Zahl(en-Darstellung)
  - Objekt
  - Applet
  - Methode
  - Exception
  - Thread
  - http und HTML, XML

Literatur

C:

- Programmieren in C, Kerninghan & Richie, Hanser Verlag
- Die Programmiersprache C, Nachschlagewerk des RRZN Hannover
- Das C Grundlagenbuch, Gerhard Willms, Data Becker. (incl. CD mit C Compiler)
- Teach yourself C in 24 hours, Tony Zhang, Sams Publishing.(incl. CD mit C Compiler)
- online- Hilfe der jeweiligen Entwicklungs- Umgebung

Java:

- Java Development Kit (JDK)
- Java in a Nutshell (incl. Beispiel-Band) , David Flanagan, O'Reilly
- Java 1.1 Schnellübersicht, Volker Schmidt, Markt & Technik
- Not just Java, Peter van Linden, Prentice Hall
- Java in 21 Tagen, Laura Lemay, Sams Publishing
- Java, Nachschlagewerk des RRZN Hannover

HTML:

- Selfhtml, Stefan Münz
- viele Bücher in fast allen Buchhandlungen

Verbreitete Programmier(hoch)sprachen

Basic:

- leicht erlernbare Programmiersprache.
- stößt rasch an seine Grenzen
- oft verwendet für „Quick & Dirty“ Programme
- wird meistens als Interpreter*) Sprache implementiert.

Visual-Basic:

- Microsoft Variante von Basic mit der Möglichkeit Windows-Programme zu schreiben
- Nur für WINTEL-Plattformen verfügbar und sinnvoll.
- Setzt Verständnis für die Windows Programmierung voraus.

FORTRAN:

- Eine der ältesten Hochsprachen
- Weitverbreitet bei naturwissenschaftlichen Anwendungen

COBOL:

- War sehr verbreitet bei „kommerziellen, betriebswirtschaftlichen“ Anwendungen
- Wird z. Zt. vielfach von proprietären Sprachen z.B. ABAP/4 (SAP) verdrängt
- Seit dem Jahrtausendwechsel nicht mehr sehr weit verbreitet und vom Aussterben bedroht.

Pascal:

- Programmiersprache von N. Wirth an der ETH Zürich für die Lehre entwickelt
- wird auch kommerziell verwendet
- vor allem von Borland als kommerzielles Produkt weiterentwickelt und vertrieben.(Delphi)

C[*)]:

- Sprache wurde ursprünglich zum Schreiben eines plattform-unabhängigen Betriebssystems (Unix) entwickelt. C stellt gewissermaßen die Fortsetzung von Assembler[*)] mit anderen Mitteln dar.
- Die ersten Hardware-Plattformen auf denen C implementiert wurde waren die PDP-Rechner der Fa. Digital. So findet man für fast alle Assembler/Maschinencode Befehle der PDP 11 ein direktes Äquivalent in C. z.B.: "INC" --> "++" oder der Autoinkrement Befehl --> "A++"
- Alles was technisch möglich ist, lässt sich – im Gegensatz zu Pascal - in dieser Sprache formulieren.
- Mit Pointern kann man auf alles und jedes zugreifen.
- Die Devise[*)] von C lautet: „Trust the Programmer“.
- Es gibt kaum Möglichkeiten den Programmierer vor sich selbst zu schützen.
- Aufgrund der inzwischen sehr ausgereiften Compiler ist der erzeugte Maschinencode sehr schnell und/oder kompakt – je nach Compiler Einstellung.

- C wird gerne für hardwarenahe Programmierung – z.B. von embedded Systemen – verwendet.
- C kann durch Bibliotheken beliebig erweitert werden.
- Preprocessor nicht mehr zeitgemäß.
- Elementgröße der Datentypen (z.B. von int) ist nicht normiert. int ist manchmal eine 16-Bit- (DOS) und manchmal 32-Bit- (Windows) Variable. --> Probleme bei der Portierung von Programmen.

C++:

- Objektorientierte Erweiterung von C. (Hieß ursprünglich „C with classes“).
- Komplexer als C und Java – speziell wegen der unnatürlichen Mehrfachvererbung.
- Da jedes gültige C Programm auch als C++ Programm durchgeht, erbt auch C++ die „Sünden“ von C.
- Wegen der uneingeschränkten Pointerverwendung ist kein Garbage Collect möglich. --> Fast alle größeren Programme weisen sog. „Memory Leaks“ auf.
- wie C auf das Prinzip "Trust the Programmer" festgelegt.
  ("In C ist es leicht sich in den Fuß zu schießen. In C++ ist es schwieriger, wenn man es doch tut, reißt es einem gleich das ganze Bein ab!" B. Stroustroup, C++ Erfinder)
- C++ wird oft zusammen mit Klassenbibliotheken wie MFC eingesetzt. (Nur mit deren Hilfe lassen sich Windows Anwendungen mit vernünftigem Aufwand realisieren.)

Perl:

- Interpretierende Sprache mit starken Text- und Stringbearbeitungsmöglichkeiten.
- Syntax etwas kryptisch
- Erlaubt einen „hemdsärmeligen“ Programmierstil.
- Enthält auch Methoden zur Netz- (Internet-) Programmierung.
- Wird sehr häufig für die Programmierung von Web-Applikationen (CGI-Scripten) verwendet.
- Inzwischen um ein Toolkit (Tk) erweitert zum Erstellen von grafischen User Interfaces.
- Der Apache Web-Server wird inzwischen auch mit einem Perl-Modul (mod_perl) ausgeliefert.

Java:

- neue Programmiersprache mit der Syntax von C
- ohne die Altlasten von C , die in seiner Historie begründet sind.[C++-]
- 100% Objekt-orientiert, deshalb auch als Einstieg in die OO-Denkweise geeignet
- Plattform unabhängiger (Byte-)Code, der auf allen Rechner ohne Modifikation ausführbar ist.
- Geeignet für sog. "Enterprise Applications"
- Netz- (Internet-) Funktionalität im Sprachumfang enthalten
- Inzwischen sind Java-basierende Mikroprozessoren verfügbar, die den Java Bytecode als „native instruction set“ enthalten und damit die schnelle Ausführung von Javaprogrammen ermöglichen.

ABAP:

- proprietäre Sprache der Fa. SAP, Walldorf.
- ursprünglich zum Erstellen von Reports gedacht.
- wird häufig als Sprache zur Realisierung von beliebigen Applikationen in SAP verwendet
- verdrängt zunehmend COBOL
- weist Ansätze von Objekt-Orientierung auf.

Scriptsprachen:

- Javascript
- VBScript
- MS Scripting Host
- REXX
- DOS-Batch
- tcl/tk
- DCL

  ....und sehr viele proprietäre Sprachen. Viele große Programmsysteme wie Excel, Word, Documentum etc.haben eigene Script- oder Macro-Sprachen.

Jede (Hoch-)Sprache hat Ihre besonderen Vor- und Nachteile. Es gibt bei Programmiersprachen ebenso wenig eine „bessere“ oder „schlechtere“ Sprache wie bei natürlichen Sprachen. Unterschiede liegen oft weniger in Ihrer Eignung für bestimmte Aufgaben, als in der Gewöhnung der Benutzer. (z.B. Firmenstandards, persönliche Präferenzen, Vorhandensein von Entwicklungstools etc.)

***) Assembler:**

- Assembler-*Sprache* ist eine maschinennahe und damit maschinen-abhängige Sprache.
- Jedem Assembler Befehl entspricht eine Instruktion auf der Maschinenebene.
- Das Assembler-*Programm* ersetzt lediglich die mnemonischen Befehle wie „mov“ durch ihr numerisches (hexadezimales) Equivalent auf der Maschinenebene.
- Die meisten Assembler erlauben die Verwendung von „Macros“ und symbolischen Namen.

***) Disassembler:** Es gibt Programme sog. Disassembler, die aus den Maschinenbefehlen wieder die mnemonischen Assembler Befehle rekonstruieren. Disassembler kehren den Prozess des Assemblierens um. d.h. sie versuchen aus dem Maschinencode den Assemblercode zu rekonstruieren (Reverse Engineering). Sie sind oft Bestandteil von Debuggern und Entwicklungstools.

Anm.: Das was bei Assembler-Programmen - bis auf die Kommentare, symbolische Namen und die Code-Strukturierung (z.B. Einrückungen, Macros) - noch einigermaßen gelingt, schlägt bei Hochsprachen in der Regel fehl. Eine Rekonstruktion des ursprünglichen Quellcodes in C oder Java ist nicht möglich, weil es keine eindeutige Beziehung zwischen dem Hochsprachenprogramm (Quellcode) und dem Assemblercode gibt. ("Rekonstruktion der Sau aus der Wurst")

*) Spirit of C:

Der Geist von C ("spirit of C") lässt sich in folgende Sätzen fassen:

- Trust the programmer
- Don't prevent the programmer from doing what needs to be done
- Keep the language small and simple
- Make it fast, even if it is not guaranteed to be portable.

*) Compiler und Interpreter:

„Echte“ Compiler erzeugen sog. Maschinencode. Dieser wird zur Programmausführung dann (nur noch) in den Arbeitsspeicher geladen und nach und nach abgearbeitet. Dazu wird der Maschinencode – der aus Instruktionen und Daten besteht (von Neumann Architektur) – in die CPU geholt (fetch) und ausgeführt (execute). Maschinencode ist nicht „human readable“.
Der Vorgang des Kompilierens (=Umwandelns) und der Ausführung sind zeitlich getrennt.
Interpreter dagegen lesen das auszuführende Programm als Quellcode ein und wandeln es zur Laufzeit in Maschinenbefehle um. Da bei jedem Programmlauf eine Umwandlung stattfindet, sind Programme die interpretiert werden langsamer als solche die nur ausgeführt werden.
Die zur Zeit sehr populäre Sprache Java ist eine Mischung von Compiler und Interpreter. Bei Java erzeugt ein Java*compile*r (javac) einen sog. Bytecode. Dieser Code wird dann zur Laufzeit vom Java*interpreter* (java) eingelesen und abgearbeitet. Auch für Java gibt es Disassembler, die versuchen aus den class-Files den Javacode zu rekonstruieren.

*) Compiler, Linker, Loader

Bei Pascal, C, C++ muß der vom Programmierer geschriebene Code zunächst in Maschinenbefehle umgesetzt werden. Dazu dient ein Programm das Compiler genannt wird. Am Ende dieses Prozesses steht der sog. Objectcode. Darin sind alle Adressen relativ zu einer bestimmten Startadresse. Das Porgramm ist in dieser Phase noch nicht ausführbar. Es fehlen nämlich noch die bereits komplieren Module aus anderen Dateien und die Bibliotheken.
In einem weiteren Schritt werden die Module (im Objectcode) zu einem ausführbaren Programm zusammnen gebunden (gelinkt). Jetzt sind alle Adressen bekannt und fest eingesetzt oder sie sind relativ zu einer einzigen Startadresse. Beim Laden des Programms in den Speicher werden dann diese relativen Bezüge auch noch aufgelöst und das Programm kann ausgeführt werden.

*) Funktionen und Macros:

Funktionen, Routinen und Prozeduren im Sinne von Programmiersprachen sind in sich abgeschlossene Programmteile, die mit einem bestimmten Satz von Eingangsgrößen eine Verarbeitung durchführen und einen Satz von Ausgangsgrößen "berechnen" und für eine weitere Verarbeitung bereitstellen. Funktionen und Prozeduren werden oft in Bibliotheken zusammengefasst - und ggfs. einem breiten Publikum zur Verfügung gestellt. z.B. Mathematische Funktionen wie sin, cos etc. oder I/O-Prozeduren.

Beim Aufruf einer Funktion (call) werden entweder die Eingangsparameter direkt auf den Stack kopiert (Call by value) oder es werden Verweise (Referenzen) auf die Parameter dem Programm übergeben (call by reference). Danach verzweigt das Programm in den eigentlichen Funktionscode. Am Ende werden die Ergebnisse in einen Datenbereich des aufrufenden Programms kopiert, der Funktionswert (return code) bestimmt und die Programmausführung mit einem Rücksprung (ret) ins aufrufende Programm beendet.

Manchmal sind jedoch Funktionen so kurz, dass es sich nicht lohnen würden den gesamten o.g. Aufwand zu betreiben. In solchen Fällen kopiert der Compiler den Funktionsrumpf in das aufrufende Programm. (Macros oder inline functions). Für den Programmierer ist das allerdings transparent/unsichtbar.

Wie entsteht ein Programm (Program Lifecycle)

1. Durch eine zündende Idee
2. Durch Nachdenken über Verbesserungsmöglichkeiten von bestehenden Lösungen
3. Durch Diskussion mit dem Auftraggeber (Fachabteilung)
4. Darstellung der (Programm-)Abläufe
5. Editieren / Codieren (meist in einer Entwicklungsumgebung IDE)
6. Compilieren*)
7. Linken (statisch) mit anderen *.obj oder *.lib Dateien

   Compilieren + Linken werden oft auch als "Make" oder "Build" bezeichnet.

   (Dieser Schritt entfällt bei Programmen, die interpretiert werden wie Basic und Java.)
8. Ausführen. Falls Fehler auftreten: gehe nach 5. oder 2.
9. Abnahme durch den Auftraggeber
10. Produktion = Nutzung
11. Entsorgung d.h. Ersatz durch ein anderes Produkt oder Wegfall der Nutzungsmöglichkeiten.

Tools einer Integrierten Entwicklungsumgebung (IDE)

- Projekt-Management, Versionsverwaltung
- Text-Editor (mit Syntax Coloring)
- Integrierte Hilfe-Funktion (Kontext-sensitiv)
- Compiler*), Linker*)
- Ressource-Editor für die Oberflächen Erstellung
- Browser für Objekte und Variablen
- Netzwerk/Internet-Support
- Debugger mit folgenden typischen Funktionen:
  - Break Point und Lesezeichen setzen
  - Run until..; Step in; Step out; Step over
  - Direkte Variable Auswertung und Änderung
  - Watch Window
  - Call Stack
- Tools wie Spy, Tracer, WinDiff etc.
- u.a.

Funktions-orientierte Programme:

Programme (=Prozeduren) sind vergleichbar mit (Koch-)Rezepten:

Am Anfang stehen normalerweise die Zutaten (=Daten). Danach folgen die Aktionen, die auf diese Zutaten angewandt werden. Auch Rezepte sind auch modular aufgebaut. Bestimmte immer wiederkehrende Aktionen (z.B. brate<objekt>) werden zusammengefasst und können auf die verschiedensten – aber nicht auf alle - Zutaten und Zwischenprodukte angewandt werden.

- Programme bestehen aus Daten und Aktionen
- Im Deklarationsteil werden die Daten und ihr zugehöriger Typ erzeugt (d.h. im Speicher angelegt). Nicht immer erhalten neu angelegte Variablen einen definierten Wert.
- Die Aktionen verändern die vorhandenen Daten und erzeugen neue.
- Aktionen lassen sich zu Funktionen[*)], Modulen und Paketen bündeln.
- Im Mittelpunkt steht immer die Aktion. Der Programmierer ist dafür verantwortlich, dass die Aktion auch auf ein geeignetes Objekt angewandt wird. Er muss durch entsprechende Maßnahmen verhindern, dass z.B. die Prozedur *„brate“* auf Objekte wie „*Nachtisch*“ oder *„Milch“* angewandt wird. Diese Prüfung ist u.U. eine heikle Sache. (z.B. Objekt = *„Apfel“*)

Was passiert beim Übergang zur objekt-orientierten Programmierung?

- Datenstrukturen werden zu Klassen und Objekten (Objekte sind Instanzen einer bestimmten Klasse)
- Funktionen werden zu Methoden, die an bestimmte Objekte gebunden sind.
- Objekte werden mit Korn- bzw. De-struktoren erzeugt bzw. entsorgt.
- Funktionen werden abhängig von der Parameterliste. z.B. abs(z) <-> abs (x)
- Die Wiederverwendbarkeit von Code ist ein weiteres Ziel der OO-Programmierung.
- Vererbungsprinzip erlaubt das Erben von Eigenschaften (Methoden und Variablen) aus Vaterklassen (bei C++: Mehrfach Vererbung z.B. Amphibienfahrzeug erbt von der Klasse Schiff und von der Klasse Auto). Das Vererbungsprinzip hat kein Pendant in C.
- Gültigkeits- (Visibility-) Regeln werden zu Access-Regeln (friend-Klassen).

Die Einteilung in Klassen und die Verteilung der Methoden auf die Klassen ist keine leichte Aufgabe und erfordert ein tiefes Systemverständnis. Der viel beschworene „Code Re-use“ scheitert in der Praxis meist an einer nicht übertragbaren bzw. schlecht überlegten Klasseneinteilung.

Mein 1. Javaprogramm

```
/*************************************************************************
  file   : hello.java

  Autor    : K.O. Linn
  Version  :   1.0 vom 22.9.98

  Calling Seq.:
        hello


  Beschr.:
      Gibt den Satz "Hello world!" auf dem Bildschirm aus.
*************************************************Infoblock-Ende*/


public class hello {  //dieser Name muss mit dem Dateinamen übereinstimmen
    public static void main(String args[]) {  //Klassen Methode
        System.out.println("Hello world!"); //zeilenweise Ausgabe
        return;
    }
}
```

Programmelemente

1. Zeichensatz [Kap. 5.1]
2. Schlüsselworte [Kap. 5.5]
3. Kommentar(e) [Kap. 5.3]
4. Datentypen [Kap. 8]
    a. generische Datentypen [Kap. 8.1]
    b. zusammen gesetzte Datentypen (C: struct, union)
    c. Objekte [Kap. 6]
5. Operatoren [Kap. 5.4]
6. Kontrollstrukturen [Kap. 9]
    a. Block [Kap. 9.1]
    b. Schleifen [Kap. 9.3]
    c. Bedingungen [Kap. 9.2]
    d. Sprunganweisungen [Kap. 9.4]
7. Exceptions [Kap. 10]
8. Threads [Kap. 11]

Ausnahme Behandlung (Exception Handling)

Die meisten Programme der funktionieren nicht immer so wie vom Erfinder geplant. So werden z.B. Dateien, die gelesen werden sollen nicht gefunden, Array-Elemente sollen verarbeitet werden, sind aber überhaupt nicht vorhanden usw.
Die Behandlung dieser Ausnahmefälle ist meist aufwendiger als das eigentliche Programm selbst. Java bietet – wie andere OO-Sprachen auch – in dieser Hinsicht einige Vorteile gegenüber der in C verwendeten Technik.
In Java ist es möglich ganze Programmblöcke mittels des Konstruktes
try {...} catch() {....} catch () {...} .... finally {...} gegen unliebsame Überraschungen abzusichern und damit robuster zu machen. Die Vererbung tut ein Übriges um Rechnerabstürze aufgrund von Fehlern in einem einzelnen Programm zu verhindern. Normalerweise kümmert sich ein Programmierer nur um die Ausnahmen, die er selbst geschrieben hat oder deren Verhalten er verändern will. Alle anderen überlässt er sich selbst d.h. den Vererbungsregeln der Objekthierarchie. Ausnahmen die nicht sofort abgefangen werden, wandern die Objekthierarchie aufwärts, bis sie irgendwann behandelt werden. Spätestens auf der Ebene „Object“ – der höchsten in der Javaobjekthierarchie – werden die letzten abgefangen. Allerdings werden Reaktionsmöglichkeiten immer allgemeiner und wirkungsloser, je weiter man in der Hierarchie aufwärts steigt. Am Ende bleibt nur noch das Ausdrucken eines Fehlertextes und des Call-Stacks.

User Sicht:

Der fragliche Programmblock wird in ein „try“-Statement eingebettet. Die potentiellen Fehler, die innerhalb dieses Block erwartet werden, werden einzeln oder gesammelt in einem „catch“-Block behandelt. In einem „finally“-Block können gemeinsame Aufräumungsarbeiten durchgeführt werden. (Für diesen Fall werden in C-Programmen gerne goto-Statements verwendet, die es ja in Java nicht gibt.)

Erzeuger Sicht:

Exceptions sind ebenfalls Objekte, die alle von der Klasse „Throwable“ abgeleitet werden.
Wenn eine Methode in eine Situation gerät aus der sie sich nicht mehr selbst befreien kann, muss sie eine Exception generieren mit Statements wie:

```
throw new myException1 („Die Datei konnte gefunden werden!“);
```

Die Methode in der das Ereignis auftreten kann muss dies mit Worten wie:

```
public void start throws myException1 {
```

kund tun.

Parallelverarbeitung (Threads)

In bestimmten Situationen möchte man mehrere Funktionen nicht wie üblich nacheinander sondern gleichzeitig ausführen. Ein gutes Programm sollte nämlich auch dann „reaktiv“ bleiben, wenn es gerade beschäftigt ist und in der Lage sein eine Bildschirmausgabe zu erzeugen während es auf die Tastatureingabe wartet. Zur Lösung dieses Problems kann man entweder in seinem Programm entsprechende Maßnahmen ergreifen oder auf die Unterstützung des Betriebssystems zurückgreifen.

- **Eigene Maßnahmen**: Man konstruiere einen – und nur einen – „Eventloop“ und sorge dafür dass das Programm nirgendwo in dieser Schleife hängen bleiben kann,
  in dem man sich, vor jedem Programmschritt der potentiell lange dauern kann, versichert, dass er in kurzer Zeit abgeschlossen werden kann. z.B. immer `kbhit()` vor `getchar()`verwendet.
- **BS Unterstützung:** Man zerlegt den Programmablauf in mehrere Handlungsfäden (Threads) und überlässt dem BS den Rest. Das o.g. Problem würde 2 Threads erfordern: einen für die Bildschirmausgabe und einen für die Tasturabfrage.

Bei der Verwendung von Threads wird fast immer der Einsatz von Synchronisationsmechnismen erforderlich. Dazu gibt es verschiedene Konstrukte die eine Synchronisation von verschiedenen Threads ermöglichen.(Semaphoren/Flags, critical sections, Mutexe u.a.)

C2.doc