```
class StudentischeGemeinschaft {
        public static void main(String[] args) {
          Mensch Maggie;
          if (Maggie.writesSkript() == true) {
             Maggie.getSkript = false } } }
```

1. Theoretische Grundlagen

1.1 Softwarequalität

(A) Externe Qualitätskriterien (benutzer-/ anwenderorientiert)

Korrektheit: Funktionen müssen Anforderungen entsprechen; Robustheit: unter unnormalen Bedingungen arbeitet die Software sicher; Erweiterbarkeit; Wiederverwendbarkeit; Integrierbarkeit: Möglichkeit der Kombination mit anderen SW-Produkten; Effizienz: optimales Ausnutzen der vorhandenen Ressourcen; Portabilität; Verifizierbarkeit: kann das SW-Produkt geprüft werden; Integrität: wie geschützt, wie wartbar; ease-of-use: Bedienbarkeit; Wartbarkeit

Bestandteile der Wartung: 41,8% Änderungen in den Benutzungsanforderungen, 17,4% Änderungen in den Datenformaten, 21,4% Fehlerbehebung, 6,2% Hardwareänderungen, 5,5% Dokumentation, 4% Systemtuning, 3,4% Sonstiges

(B) interne Qualitätskritierien (programmierer- / entwurtsorientiert)

1. Modularität; 2. interne Dokumentation (UML); 3. Übersichtlichkeit, 4. Lesbarkeit; 5. Design Patterns (Entwurfsmuster); 6. strukturierter Aufbau

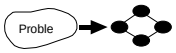
Verbesserung der externen Qualität nur durch Verbesserung der internen Qualität

1.2 Modularität

modular: 1. Angabe von Modularitätskriterien (Kriterien, über Qualität der Entwurfs-methode); 2. Angabe von Mod'prinzipien (ihre Einhaltung hilft, die Mod'kriterien zu erfüllen); 3. Mod. entspricht Architektur der SW

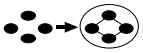
1.2.1 Modularitätskriterien

1. Zerlegbarkeit: Zerlegung eines Problems in mehrere, überschaubare Einheiten; Reduzierung der Komplexität; iterativer Prozeß.



Top-down Methode

2. Zusammensetzbarkeit: Kombination von SW-Modulen, um neue SW-Produkte zu erzeugen


Bibliotheken, Unix-Pipe Mechanismus

3. Verständlichkeit: Leser braucht wenig Wissen über Nachbarmodule, um Funktionsweise zu verstehen.

4. Kontinuität: geringe Änderungen bezüglich Anforderungen haben nur Änderungen in wenigen Modulen zur Folge - Architektur bleibt stabil

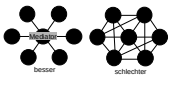
5. Schutz / Abschottung: 1. Fehler sollten auf das Modul beschränkt sein; 2. keine Fehlerpropagation

1.2.2 Prinzipien der Modularisierung

(1) Linguistische Einheiten: 1. Modul muß einer syntaktischen Einheit einer Programmiersprache entsprechen; 2. separate Kompilierung muß möglich sein
Unterstützt werden: Zerlegbarkeit; Zusammensetzbarkeit; (Schutz/Abschottung)

(2) Wenig Schnittstellen zu anderen Modulen:
Jedes Modul sollte mit möglichst wenig anderen Modulen kommunizieren.
Unterstützt werden: Kontinuität; Zusammensetzbarkeit; Schutz

(3) weak coupling: falls zwei Module miteinander kommunizieren, dann sollen diese so wenig Informationen wie nötig austauschen (begrenzte Bandbreite) *Unterstützt werden:* Kontinuität; Schutz



(4) Explizite Schnittstellen: Falls Modul A mit B kommuniziert, so muß man dies in beiden Quelltexten erkennen können. Direkte Manipulation von Daten ist verboten! *Unterstützt werden:* Schutz; Kontinuität

(5) Information hiding: alle Daten/Informationen eines Moduls sind privat => die Leistungen des Moduls sind nur über seine expliziten Schnittstellen der Außenwelt bekannt.=> Kapselung von Daten und Wissen
Unterstützt wird: Kontinuität

Zusammenfassung:

"Bei der Modularisierung wird ein totalitäres Regime in der Gesellschaft der Module eingeführt. Jeder soll mit möglichst wenig anderen Modulen sprechen und wenn kommuniziert wird, dann mit möglichst wenig Worten. Wenn etwas gesagt wird, dann geschieht dies öffentlich und laut."

1. Modularität ist Schlüssel für die Erreichung der Qualitätskriterien Wieder-verwendbarkeit und Erweiterbarkeit; 2. modulare Konzepte müssen in Spezifikation, Design und Programmierung beachtet werden; 3. Art der Kommunikation ist wichtig für Erhaltung modulare Architekturen; 4. nur durch Information Hiding kann eine langfristige Integrität des Moduls erreicht werden

1.3 Ansätze zur Wiederverwendbarkeit

Ziel: Verwendung von Softwaremodulen ähnlich der VLSI-Bausteine bei der Hardwareentwicklung

Probleme: 1. Informationsproblem/unbekannte Lösungen (die Existenz von Lösungen bzw. Modulen muß bekannt sein); 2. Programmierer wollen alles selber machen - immer kleine Unterschiede

Bisherige Ansätze: 1. Wiederverwendung von Source Code (-> teuer, im kommerziellen Bereich schwierig); 2. Wiederverwendung von Personal; 3. Wiederverwendung von Entwürfen

1.3.1. Bibliotheken - Sammlung von Funktionen,Unterprogrammen

Einsatz wenn: 1. Funktionen unabhängig voneinander; 2. keine komplizierte Datenstrukturen; 3. relativ wenig Parameter

Bsp.: Wie kann man das Problem der Generizität mit Hilfe von Funktionen lösen? Wie kann man Kellerspeicher für unterschiedliche Datentypen realisieren?

1. Lösung: Duplizieren des Quelltextes, Abändern des Datentyps (=> sehr viele ähnliche Funktionen, sehr korrekturunfreundlich)

2. Lösung: eine Funktion mit vielen Parametern (=> switch in der Funktion, neu kompilieren bei Erweiterung, Problem der Instantiierung)

Hauptkritik: Funktionen, nicht die Daten stehen hier im Vordergrund

1.3.2 Der objektorientierte Ansatz

Objekt: 1. beschreibt einen Ausschnit aus der realen Welt oder eines Modells; 2. besitzt Merkmale (Eigenschaften, Attribute) und Operationen (Funktionen, Methoden, Botschaften); 3. Merkmalswerte bestimmen Objektzustand; 4. nur durch Methoden kann O-Zustand ermittelt/verändert werden.

Klasse: 1. beschreibt Eigenschaften durch Merkmale und Methoden; 2. Muster von gleichartigen Objekten.

Exemplare, Instanzen: 1. können aus einer Klasse erzeugt werden; 2. besitzen Merkmale sowie Methoden ihrer Klasse => Schaffung eines neuen Datentyps

Problem: 1. wie findet man die Objekte; 2. bei der OO werden die äußeren Funktionen so spät wie möglich realisiert; 3. Hauptgewicht besteht in Analyse der Klassen; 4. Design besteht in Verbesserung der Klassen; 5. Wiederverwendbarkeit und Änderbarkeit sind die wichtigsten Design-Kriterien

Abstrakter Datentyp (ADT): beschreibt eine Klase nicht durch Implementierung, sondern durch Kombin. anderer Klassenmethoden

OO-Systemdesign: Konstruktion eines Systems nur aus ADT's

Eigenschaften eines OO-Systemsdesigns nach Meyer

1. Object-based-modular-structure (nur Objekte); 2. Data-abstraction (ADT); 3. automatic-memory management (garbage collector, muß nicht realisiert sein); 4. classes (Klassen); 5. inheritance (Vererbung); 6. Polymorphismus & dynamic binding (Zuweisung von Childreferenz an Parent, poly morph=viele Gestalt); 7. multiple inheritance (mehrere Elternklassen)

2. Klassen

2.1. Einführung

Ziel: Sprachkonstrukt, um ADT's zu definieren und wie Basistypen zu behandeln, z.B. Datentyp Complex mit Zugriffsfunktion der Merkmale

Vorteile: 1. Programme, deren Datentpyen der Anwendung ungefähr entsprechen, leichter zu verstehen/leichter zu modifizieren; 2. sinnvoll gewählte Menge benutzerdefinierter Datentypen macht Programm klarer; 3. Compiler kann illegale Verwendung von Objekten entdecken (andernfalls erst später beim Testen)

Grundidee: 1. Trennung der Implementierung von Eigenschaften, die für seine korrekte Verwendung wichtig; 2. Schutz der klasseninternen Datenstrukturen

2.2 Definition von Klassen

Klassen: benutzerdefinierte, zusammengesetzte abstrakte Datentypen, welche Daten + Funktionen zusammenfassen

1. Realisierung eines Datentyps Datum als struct:

```
struct Datum  { int tag, monat, jahr; };
```

2. Definition von Funktionen für die Manipulation dieser Daten

```
void setze_datum (Datum *, int, int, int);
void naechstes_datum ( Datum*);
void drucke_datum (Datum*);
```

3. Nachteil: keine explizite Verbindung zwischen Daten und Funktionen

4. Deklaration des neuen Datentyps Datum als Klasse (in Headerdatei)

```
class Datum {
        int tag, monat, jahr;
        public:
        void setze (int, int, int, int);
        void naechstes ();
        void drucke ();
        void lese (int*, int*, int*);}
```

5. Deklaration eines neuen Datentyps Complex als Klasse

```
class Complex {
        double re, im;
        public:
        double get_re();
        void   set_re(double r);
        double get im();
        void   setim (double i);
        void   get_im();
        void   drucke();}
```

wichtig: Klassen sind Muster, wie diese Objekte aussehen, bei der Klassen-deklaration werden wie bei structs auch noch keine reale Speicherobjekte erzeugt.

Allgemeine Beschreibung der Klassendefinition:

Syntax: `class + <Klassenname>`

Zugriff auf die Daten einer Klasse: die Daten einer Klasse sowie die Funktionen sind immer privat, d.h. ein Zugriff mit **->** oder **.** ist nicht möglich => explizite Kennzeichnung duch private:

public: Daten und Funktionen werden öffentlich. private: und public: können beliebig oft in einer Klassendefinition vorkommen.

Rumpf: 1. besteht aus aus Schnittstellenfunktionen, sogenannte Methoden, Botschaften (oder Memberfunktionen) der Klasse, welche die Schnittstelle zu den Objektdaten bildet; 2. Zugriff auf Objektdaten ist nur über Schnittstellen-Funktionen möglich, andere Funktionen haben keinen Zugriff - im Normalfall werden in der Klassendeklaration nur die Funktionsprotoypen definiert, eigentliche Funktions-definition an einer anderen Stelle im Programm.

Frage: Wo müssen Funktionen in C deklariert werden? In C werden alle Funktionen parallel zu main definiert.

Vorteile des OO-Ansatzes:

1. Fehlerlokalisierung wird wesentlich einfacher; 2. Nutzer des Datentyps sieht die Schnittstelle i.d Klassendefinition; 3. kein Wissen über Realisierung nötig

2.3. Definition von Memberfunktionen

(1) INLINE Funktionen

Kennzeichen: Der Funktionsrumpf wird direkt in der Klasse (*.h-File) definiert

Vorteil: Vermeidung von Funktionsverlust, da C++ anstelle eines Funktionsaufrufs den Code dupliziert

Nachteil: ineffizient, Implementierung wird sichtbar

Regel: Inline-Funktionen nur in "kleinen" Funktionen verwenden

(2) externe Funktionen

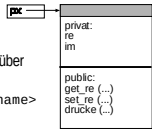
Kennzeichen: Funktionsrumpf steht außerhalb der Klassendefinition wie eine normale C-Funktion.

Syntax: <Datentyp des Ergebnisses> <Klassenname> :: <Funktionsname> (<Parameterliste>) **{**<Funktionsrumpf>**}**

Da Gültigkeitsbereich dieser Memberfunktion nicht im Hauptprogramm sondern innerhalb der Klasse, muß dies durch die Angabe **<Klassenname>::** näher spezifiert werden, z.B. externe Funktion für Datum

```
void Datum::setze (int t, int m, int j)
        { tag = t; monat = m; jahr = j; }
```

2.4 Generieren von realen Speicherobjekten (Instanzen, Exemplare)



Regel: Genauso wie jede andere Variable eines existierenden Datentyps, z.B. `Complex x;`

Zugriff auf öffentliche Datenelemente und Memberfunktionen über **.**-Notation

Syntax: `<Klassenname>.<Variablen-/Methodenname>`

2.5 Dynamische Objekte (auf dem Heap)

1. Generieren, z.B.

```
Complex *px
px = new Complex
```

2. Löschen (Speicher freigeben)

```
delete px; (free in C)
weitere Beispiele:
(*px).set_re(88.3); // oder: px->set_re(88.3);
```

2.6 Konstruktoren

Funktion setze_datum() dient der Initialisierung von Objekten

Nachteil: 1. Programmierer muß setze-Methode explizit aufrufen - wenn nicht -> Fehlerquelle; 2. kein Initialisierungszwang

Konstruktor: 1. Methode, um Objekte bei der Definition zu initialisieren; 2. hat gleichen Namen wie Klassen; 3. eine Klasse kann mehr als einen Konstruktor besitzen

(1) Klasse Datum

```
class Datum
        { int tag, monat, jahr; ...
        public:
        Datum ();
        Datum (int, int, int); // tag = ...
        Datum (char*); // Datum liegt in Textform vor
        ...}
```

Unterschiede zu normalen Methoden: können nicht explizit aufgerufen werden, haben keinen Rückgabewert.

```
Datum d;
Datum d1 (12, 3, 1999)
Datum d2("15.Okt.1999);
```

(2) Klasse complex

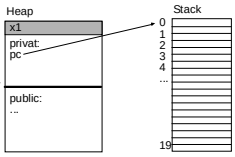
```
class complex
        { double re, im;
        public:
        //Defaultwerte
        Complex (double re = 0.0, double i=0,0)
           {re=r; im=i;}
}
```

2.7 Destruktoren

1. Methode, die aufgerufen wird, wenn Objekt vernichtet wird; 2. Benutzer kann (muß) Destruktor definieren; 3. Namenskonvention `~<Klassenname>();`4. hat keine Parameter und keinen Return-Wert; 5.es kann nur einen geben

Anwendungsbsp.: 1. Konstruktor reserviert Speicherplatz; 2.Destruktor muß diesen Speicherplatz freigeben

```
class X {
        char * pc;
        public:
        X (int l=10)
           {pc=new char (l+1);}
        ~X() {delete pc;}
}
X x1(20);
```



3. This-Variable, Friends

In einer Memberfunktion (und nur in dieser) kann auf die Elemente (Daten und Funktionen) einer Instanz zugegriffen werden.

Wichtig: Jeder Methode wird beim Aufruf ein verborgenes Argument vom Typ <Klassenname>* mit dem Namen this übergeben.

3.2 Statische Klassenelemente

1. eine Klasse ist ein Datentyp, kein Datenobjekt

2. jedes Objekt (Instanz) einer Klasse besitzt eine Kopie der Datenelemente einer Klasse

3. wird in einer Klasse ein Datenelement als static deklariert, so wird nur eine einzige Variable im Speicher erzeugt, die Instanzen erhalten keine Kopien, aber den Zugriff auf diese Variable

```
class Complex {
        double re,im;
        static int zaehler;
        public:
        Complex(double r=0.0, double i=0.0)
           { re=r; im=i; zaehler++ }
        ~Complex() { zaehler-- } };
```

Semantik der Variablen zaehler: zaehler wird im Hauptprogramm initialisiert mit `Complex::zaehler=0;`

Besonderheiten: static-Element ist als private deklariert: => Zugriff über Memberfunktion, z.B.

```
        a.get_zaehler(); a.set_zaehler(...);
```

Anwendungsbeispiele für static-Variable

1. Aufnahme von Daten, die allen Objekten gemeinsam sind; 2. um die Anzahl der sichtbaren, globalen Variablen/Namen zu reduzieren; 3. offensichtlich zu machen, welches statische Objekt logisch zu welcher Klasse gehört; 4. den Zugriff auf ihre Namen zu steuern

3.3 Friends

Bisher: Zugriff auf die privaten Klassenelemente haben nur Memberfunktionen

Definition: Eine Friend-Funktion f der Klasse X ist eine Funktion, welche die vollen Zugriffsrechte auf private Elemente der Klasse besitzt, obwohl sie keine Memberfunktion der Klasse ist.

```
class X
{       int i;
        friend void friend_func(X*, int);
        public:
        void member_func(int); };
void X::member_func(int a) { i=a;}
void friend_func(X *ptr, int a) { ptr->i=a; }
void main()
{       X xobj;
        xobj.member_func(6);
        friend_func(&xobj, 6); }
```

Wichtig: 1. kein this-Pointer, da keine Methode der Klasse, 2.Friend-Funktionen haben üblicherweise einen Parameter vom Typ der Klasse. Anwendungsbeispiel: Multiplikation von Vektoren mit Matrizen

(1) mit Friendfunktion

```
#define MAXSIZE 10
class Vektor {
        int v[MAXSIZE];
        friend Vektor mult(Matrix, Vektor); };
class Matrix {
        int v[MAXSIZE][MAXSIZE];
        friend Vektor mult(Maltrix, Vektor); };
Vektor mult (Matrix m, Vektor v) {
        Vektor r; int i,j;
        for (i=0; i<MAXSIZE; i++) {
           r.v[i]=0;
           for (j=0; j<MAXSIZE; j++)
              { r.v[i]= r.v[i] + m.v[i][j]*v.v[j]; }
        } return r; }
```

(2) ohne Friendfunktion

```
#define MAXSIZE 10
class Vektor {
        int v[MAXSIZE];
        public:
        void set(int i, int wert){v[i]=wert;}
        void get(int i){return v[i];} };
class Matrix {
        int v[MAXSIZE][MAXSIZE];
        void set(int i, int j, int wert) {v[i,j]=wert;}
        void get(int i, int j){return v[i,j];} };
Vektor mult (Matrix m, Vektor v) {
        Vektor r; int i,j;
        for (i=0; i<MAXSIZE; i++) {
           r.set(i,0);
           for (j=0; j<MAXSIZE; j++) {
                 int h1;
                 h1=m.get(i,j); h1=h1*v.get(j);
                 h1=h1+r.get(i); r.set(i,h1); }
        } return r; }
```

Memberfkt. einer Klasse können auch Friendfunktionen einer anderen Klasse sein!

```
class X {... void f(y *ptr); };
class Y {... friend void x::f(y *); };
```

Alle Memberfunktionen einer Klasse können mit einer einzige Definition zu friends gemacht werden.

```
class A;    //** Forward-Deklaration
class B {
        int i;
        friend A;
        void member_funcA(); };
class A {
        void friend_x1(B *ptr);
        void friend_x2(B *ptr); };
```

Wichtig: Alle in A enthaltenen Methoden sind friends von B. Beispiel: X „ist friend von" Y und Y „ist friend von" Z ==> geht nicht!!!

Anwendung von Friendfunktionen:

1. geregelter Zugriff auf private Elemente zum Zweck der Performance-Verbesserung; 2.Klasse hat die volle Kontrolle; 3. Unbedingt nötig bei Operatoroverloading!

aber: Durchlöcherung, Aufweichung des Gedankens des Information hidings aus Effizienzgründen.

Regel: Eine Klasse sollte so wenig Friends wie möglich haben!

3.4 Qualifikation von Membernamen

Explizite Unterscheidung zwischen Namen von Klassenelementen und anderen Namen.

Bei Namenskonflikten: 1. this-Variable; 2. Bereichsoperator

```
class X
{       int m;
        public:
        int readm() { return m;
             oder return this->m; oder return X::m;} };
```

Wichtig: Name, dem nur ein **::** vorausgeht, ist ein globaler Name (globale Variable).

4. Overloading, Überladen von Operatoren

4.1 Allgemeines

Overloading bedeutet die Verwendung eines Namens zum Aufruf verschiedener Funktionsimplementierungen (Unterscheidung anhand der Signaturen)

(a) implizites Overloading

```
int i,k;   i=i+k; // Integer-Addition
float x,y;  x=x+y; // Gleitkomma-Addition
```

Unterschiedliche Funktionen, aber gleicher Operator!

(b) explizites Overloading in C++

Definition, welche Bedeutung Standardoperatoren haben, wenn sie auf Objekte einer Klasse angewandt werden.

Vorteil: 1. Verwendung der natürlichen Operationen; 2. keine künstl. Namen; 3. bequeme Schreibweise

Syntax: `<operator> + <Operatorsymbol>`

```
class Complex
{       double re,im;
        Complex(double r=0.0, double i=0.0){re=r; im=i;}
        friend Complex operator +(Complex, Complex);
        friend Complex operator -(Complex, Complex);
        friend Complex operator *(Complex, Complex); };
Complex operator+(Complex c1, Complex c2)
{       Complex ergebnis;
        ergebnis.re=c1.re + c2.re;
        ergebnis.im=c1.im + c2.im;
        return ergebnis; }
Complex operator-(Complex c1, Complex c2)
{       ... } };
```

Wichtig: 1. Vorhandene Präzedenz- und Assoziativitätsregeln bleiben erhalten und können nicht geändert werden; 2. keine Veränderung der Ausdruckssyntax, d.h. z.B. „%“ hat immer 2 Operanden; 3.keine Definition neuer Operatoren möglich („**“ statt „pow“ -> a ** b wäre nicht eindeutig, kann Potenzierung [pow(a,b)] oder Multiplikation Variable mit Zeiger [a*(*b)] sein)

4.2 Binäre und unäre Operatoren

(1) binäre Operatoren

1. als friend-Funktion mit 2 Parametern via operator(a,b); 2. als Methode der Klasse mit 1 Parameter via a.operator(b)

(2) unäre Operatoren

1. als friend-Funktion mit 1 Parameter via operator(a); 2.als Methode der Klasse mit keinem Parameter via a.operator()

```
class X
{       friend X operator-(X);    // unäres Minus c= -b
        friend X operator-(X,X); // binäres Minus c=a-b
        friend X operator-();     // fehler, friend hat
                                     mind. 1 Operator
        friend X operator-(X,X,X);//fehler, friend hat
                                     keine 3 Parameter
        X* operator &();          // Adressoperator c=&b;
        X operator &(X);         // binäres UND, c=a&b,
                                     bitweises UND
        X operator &(X,X);        // fehler, keine 3 Ops.
};
```

Folgendes ist beim Overloading zu beachten:

1. Bedeutung eines benutzerdefinierten Operators nicht vordefiniert.

Falls Operator überladen: Sache des Klassenrealisators, diese mit einer sinnvollen Bedeutung zu versehen.

2. Sind Operatoren überladen, so sind Kombinationen mit anderen überladenen Operatoren gemäß den vorgegebenen Bedeutungen nicht gegeben. Operatorkombinationen können nicht mehr gelten: Die Bedeutung i++ „ist gleich“ i=i+1 kann verloren gehen.

3. Operatorfunktion muß entweder Memberfunktion oder Friendfunktion sein, die wenigstens ein Objekt der Klasse als Argument hat und ein solches Klassenargument als Ergebnis zurückliefert.

4. Eine Operatorfunktion, die als ersten Operanden einen Basistyp (int, float,..) haben soll, kann keine Memberfunktion sein. Sie muß als Friendfunktion realisiert werden. Operatorfunktionen sollten auch mit Basistypen funktionieren:

Complex a;	friend	Methode
(1) a+2;	+(a,2);	a.+(2);
(2) 2+a;	+(2,a);	2.+(a); *//geht nicht, da 2 kein Objekt!!*

5. Beim Überladen können sonst vorhandene Operatoreigenschaften (Assoziativität, Kommutativität) verloren gehen

4.3 Benutzerdefinierte Datentypkonvertierung

Ziel: Anstatt Verknüpfungen mit Operatoren komplexer Zahlen soll eine Verknüpfung mit Basistypen möglich sein.

1. Lösungsmöglichkeit: Erweiterung der Operatorfunktionen um all diese Fälle durch Friendfunktionen

```
class Complex
{       double re,im;
        public:
        Complex(double r, double i) { re=r;im=i; }
        friend Complex operator+ (Complex, Complex);
        friend Complex operator+ (double, Complex);
        friend Complex operator+ (Complex, double);
        ... };
```

Nachteil: viel Schreibaufwand, redundanter Code

2. Lösungsmöglichkeit: Deklaration eines Konstruktors, welcher aus einem double ein Klassenobjekt (Complex) aufbaut

```
class Complex
{       double re,im;
        public:
        Complex (double r, double i=0.0) { re=r;im=i; }
        friend Complex operator+ (Complex, Complex);
        friend Complex operator- (Complex, Complex);
        friend Complex operator* (Complex, Complex); };
```

Regel: Wird beim Funktionsaufruf ein bestimmter Datentyp erwartet, und er mit Hilfe eines Konstruktors erzeugt werden kann, so wird dieser verwendet.

```
a=b*2  // 2 wird automatisch umgewandelt in re=2.0, im=0.0
```

4.4 Konversionsoperatoren

Konstruktoren für die Datentypkonvertierung unterliegen den folgender Einschränkung: 1. Es besteht keine Möglichkeit, eine implizite Typkonvertierung aus einem benutzerdefinierten Datentyp in einen Basistyp durchzuführen, da die Basistypen keine Klassen sind, z.B. Complex —> int

Lösungsansatz: Mache aus einem Objekt einen Basistyp!

Syntax: `operator + Datentyp + ( —> )    // i.d.R. keine Parameter!`

```
class Complex
{       ...
        public:
        Complex(double r, double i=0.0){re=r; im=i;}
        operator int() { return re; }
        friend Complex operator+(Complex, Complex)
        ... };
Complex a(1.0,1.0); Complex b(2.0,2.0); int i; i=a+b;
```

Das Ergebnis von a+b wird reduziert und nur der Realteil i zugewiesen:

```
#include <stdio.h>
class Winzig
{       private:
        int assign (int i)
        { if (!((i>=0)&&(i<=63)))  // Wertebereich 0..63
                { printf(„Fehler:i=%d\n“,i); v=0;
                } else { printf(„Assign: %d\n“,i); v=i;}
                return v;
        } int v;
        public:
        Winzig(int i) {assign(i);}
        operator int()
          { printf(„Konvertierung nach int: %d\n“,v);
            return v;}
};
void main()
{       Winzig c1(44), c2(2), c3(66); int i;
        i=c1;  // i=44
        i=c3;  // i=0;
        i=c1+c2;  // i=46; }
```

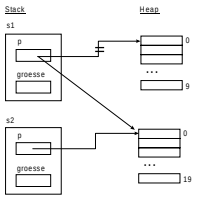
Bei Zuweisung eines Winzig-Objekts an ein int, wird eine int-Konversion durchgeführt.

4.5 Zuweisung und Initialisierung

Wichtig: `int i; int j; i=j;`

In C++ wird bei einer Zuweisung **bitweise** kopiert!

Was könnte am folgenden Beispiel falsch laufen?




```
#include<stdio.h>
class String {
        char *p;
        int groesse;
        public:
        String (int gr)
          {p=new char[groesse=gr];}
        ~String(){ delete []p; }
};
void main()
{       String s1(10), s2(20);
        s1=s2; }
```

Was macht s1=s2? Der Zeigertyp von s1 wird überschrieben und zeigt auf das gleiche Haldenobjekt wie p von s2.

Folgerung: 1. Haldenobjekt von s1 wird nicht korrekt freigegeben. 2. beim Verlassen des Blocks wird der Destruktor sowohl für s1, als auch für s2 aufgerufen. Der Heapbereich von s2 wird somit zweimal freigegeben. —> i.d.R. Programmabsturz!!!

Lösung: Überladen des Zuweisungsoperators

```
void String::operator=(String &a)
{       if(this==&a) return; //Spezialfall s1=s1
        delete[]p;
        p=new char[groesse=a.groesse]
        strcpy(p,a.p); }
```

Ein neues Haldenobjekt von der Größe des zugewiesenen Operanden wird dem 1. Operanden zugewiesen.

Keine bitweise (flach) Kopie bei s1=s2. Ganze Datenstruktur wird neu im Objekt s1 aufgebaut (tiefe Kopie).

Folge: Beim Verlassen des Blocks enstehen keine Fehler mehr, da s1 und s2 unterschiedliche p-Vektoren (char []) besitzen.

Problem: Was passiert bei Initialisierung?

```
{       String s1(10); String s2=s1; }
        // Der =Operator wird nicht aufgerufen!
```

Lösung: Für eine Initialisierung eines benutzerdefinierten Datentyps X mit einem Objekt desselben Typs sorgt ein Copy-Initialisierer mit folgendem Aussehen:

```
Klasse X => X(&X)
String::String(String &a)
{       pc=new char[groesse=a.groesse];
        strcpy(p,a.p); }
```

Ein Copy-Initialisierer wird in folgenden Situationen aufgerufen:

1. Initialisierung

```
        String s1=s2; String s3(s2);
```

2. Kopien in und aus Funktionen

Übergabe via CBV oder wenn ein Objekt als Return-Wert einer Funktion geliefert wird! z.B.

```
        String s1(10), s2(20);
        s1 = func(s2);
```

Wirkung: Copy-Konstruktor wird zweimal aufgerufen, Zuweisungs-Konstruktor wird einmal aufgerufen

4.6 Friends und Members

Wann sollen Memberfunktionen bzw. Friendfunktionen verwendet werden, um auf die privaten Daten einer Klasse zuzugreifen?

Manche Funktionen müssen Memberfunktionen sein: Konstruktoren, Destruktoren, (virtuelle Funktionen)

Hauptunterschied Member/Friends:

1. Eine Methode kann nur für ein (reales) Objekt aufgerufen werden; 2. Friend-Funktion kann auch für Basistypen benutzt werden

Regeln: 1. Eine Operation, die den Zustand eines Objekts einer Klasse ändert, sollte eine Methode sein; 2. Wird für alle Operanden einer Operation implizite Typkonvertierung gewünscht, so sollte dafür eine Friendfunktion benutzt werden.

5.1 Memberobjekte

In einer Klasse werden eines bzw. mehrere Objekte von bereits vorhandenen Klassen definiert. (Besteht aus-Relation).

UML: leere Raute = Referenz, Aggregation

```
class Auto {
        Rad r1; Rad r2; Rad r3; Rad r4; ...}
```

1. automatischer Konstruktoraufruf bei freidefinierten Objekten; 2. Memberobjekte werden nie mit einer Argumentenliste definiert; 3. Memberobjekt wird erst initialisiert, wenn Objekt der umschließenden Klasse definiert wird; 4. Konstruktoraufruf der Memberobjekte geschieht **vor** Konstruktoraufruf der umschließenden Klasse

```
Auto(float r) : r1(r), r2(r), r3(r), r4(r)
        //Doppelpunkt = expliziter Konstruktoraufruf
```

UML: volle Raute = Komposition. Alle komposit-Objekte werden automatisch mitvernichtet.

Besonders an dieser Stelle noch mal einen Dank an Margit Doelker!!!

5.2 abgeleitete Klassen

Vererbung (ist-ein Beziehung) = Erstellen von neuen Klassen aus bereits vorhandenen.

UML: leerer Pfeil von Child auf Parent(s).

Ziele: Spezialisierung, Anpassung/Erweiterung der Schnittstelle der Basisklasse, Wiederverwendbarkeit

Nachteil: nur statische Beziehung möglich

```
private class/public struct/public union + <Name der
abgeleiteten Klasse> + : + <Zugriffsrechte auf Basis-
klasse> + <Name der Basisklasse>
```

Regel: Zeiger einer Basisklasse dürfen auf Objekte dieser Klasse, sowie abgeleiteter Klassen zeigen, aber nicht umgekehrt! Gültigkeitsbereich abgeleiteter Klasse zu Basisklasse verhält sich wie innerer zu äußerer Block.

private: nur von Methoden dieser Klasse zugreifbar, children haben keinen Zugriff!

protected: wie private. children haben jedoch Zugriff

public: alle haben Zugriff

Ändern der Schnittstelle der Basisklasse: Nur public-Elemente ändern sich bei der Vererbung. Private ist der Defaultwert.

```
class unter : public ober // kein Einfluß auf Methode
class unter : protected ober
```

public Elemente der Basisklasse -> protected in der child-Klasse!!!

```
class unter : private ober
```

public Elemente der Basisklasse -> private in der child-Klasse!!!

Explizites Öffentlichmachen: `<Klassenname> + : + <Elementname>`

```
class Vorgesetzter : private Angestellter {
        public : Angestellter:nächster(); };
```

keine Neudefinition, gilt nur für public-Methoden der Basisklasse

Konstruktoren/Destruktoren: Konstruktor der Basisklasse muß! explizit von child-Klasse aufgerufen werden. Abarbeitung: 1. Konstruktor parent (wenn super in child an erster Stelle des Konstruktors), 2. Konstruktor child, ... Aufruf der Destruktoren erfolgt in umgekehrter Richtung.

```
class X {int a,b;
         public: X (int i, int j):a(i), b(j) }
         // oder X (int i, int j) { a=i; b=j; } }
```

Mehrfachvererbung und Probleme bei fehlendem Overriding:

```
class Apfel: public Baum, public Frucht { ... }
Apfel *ap = new Apfel; ap->ident(); //Eindeutigkeit nicht
gegeben.
```

Lösung: 1. Gültigkeitsoperator ap->frucht::ident(); oder 2. eigene Apfel-ident() Methode

6.Virtuelle Funktionen

Wie kann man feststellen auf welches Objekt der Zeiger vom Typ *basisklasse** zur Zeit zeigt?

1. Einführung eines Typfeldes in die Basisklasse; 2. verbieten. Nachteil: Quelltextänderungen bei neuen Klassen, viel Programmieraufwand, recompile notwendig; 3. virtual + Funktionsdefinition. Die Methode wird erst zur Laufzeit ermittelt.

Virtuelle Funktion: 1. Funktion in Basisklasse mit virtual, 2. Signaturen stimmen überein, 3. tiefste Funktion wird aufgerufen

Realisierung als Sprungtabelle, die zur Laufzeit ausgewertet wird, evtl. Performance-Verluste

abstrakte Methode: z.B. `virtual show()= 0 //Nullzuweisung erforderlich`

7. Templates

Realisierung generischer Funktionen und Klassen durch Muster in denen Datentyp/ Konstante noch variabel sind. Es entstehen Klassen-/Funktionsfamilien.

Vorteil: ein einziger Quelltext. Der Compiler erzeugt aus Beschreibung die zu Basistyp gehörende Klasse.

Wichtig: Klassendefinition und Realisierung müssen in einer Datei stehen!!!

```
template<class T>
        void swap(T &x, T &y)
          { T temp; temp=x; x=y; y=temp; } //Funktion
template<class T, int size>
        class Stack { T feld[size]; ...} //Klasse
void main() {
        Stack<int,20> sint;
        Stack<float,33> sfloat;
        sint.push(12); sfloat.push(2.33); ... }
```

JAVA

1990: Zielè Betriebsystem für Mikroprozessortechnik (Video, Telefon, VTR ....)

1. James Gosling entwickelte objektorientierte Sprache (OAK); 2. Verhandlungen mit Elektronik Herstellern scheiterten., niemand wollte OAK lizensieren; 3. 1995 aus OAK wurde Java für Internetanwendungen => Evolution statt Revolution

Ziel: einfache, bekannte Sprache somit C++ ähnlich

Robustheit und Sicherheit: 1. Java überprüft Feldindizes zur Laufzeit; 2. Keine Variablen vom Typ Zeiger, keine Zeigerarithmetik; 3. autonome Speicherfreigabe durch Garbage Collector; 4. keine Mehrfachvererbung möglich

Performance: Javainterpreter ca. 20 mal langsamer als C++. Wird aber durch Threads (parallele Prozesse) substituiert.

Portabilität: Plattformunabhängige ProgSprache, ideal für Internetanwendungen.

verteilte Anwendungen: Application, Applet(werden in HTML-Code implementiert), wird durch Webbrowser interpretiert (wird vom Client ausgeführt). SW muß nicht mehr lokal installiert sein, kann übers Netz gezogen und ausgeführt werden. immer aktuelle SW.

Java Entwicklungsumgebung

Pfad CLASSPATH muß gesetzt werden.

```
        Compiler      = javac <Dateiname>.java
          //=> plattformunabhängiger Bytecode .class
        Interpreter   = java <Dateiname>
          // Extension '.class' darf nicht angegeben
werden
        Disassembler = javap <Dateiname>
          // bringt JavaBytecode in leserliche Form
        Dokugenerator= javadoc <Dateiname>
          // erstellt aus den Kommentaren des Java-
             Quelltextes eine Doku in HTML
        Appletviewer = appletviewer
          // austesten von Applets ohne einen Webbrowser
```

Java Applets

Kleine Programmstücke, die nur in der Umgebung eines Browsers laufen.

```
import java.applet.*;
import java.awt.*;
public class HelloApp extends Applet {
        public void init()  { resize(100,100);  }
        public void paint(Graphics g) { g.
drawString("Hello World",5,20); }
}
```

init() wird immer beim Start eines Applet ausgeführt und zwar nur einmal. **start()** und **stop()** werden immer dann aufgerufen, wenn im Browser hin-und hergeblättert wird. **paint()** wird immer dann ausgeführt, wenn sich das Applet auf dem Fensterbereich darstellen soll. **update()** löscht zunächst den Bildschirm und ruft anschließend die paint()-Methode auf. Sollte durch Overriding übergangen werden. update(),

Einbindung in HTML:

```
<APPLET CODE="HELLO WORLD" WIDTH="150" HEIGHT="25">
<PARAM NAME="author" value="M. Gerstmair, S. Heinzer,
..."> ...
</APPLET>
```

Datentypen und Kontrollstrukturen

Ganze Zahlen: int (32 Bit), long (64 Bit), short (16 Bit), byte (8 Bit); per default: 0

Gleitkommazahlen: float (32 Bit), double (64 Bit); per default: 0

Zeichen: char (16 Bit); per default: '\u0000'

Logische Werte: boolean; per default: false

Referenzen: per default: **null**

Operatoren auf primitiven Datentypen: alle aus C bekannten Operatoren sind gültig!

Präzedenz und Assoziativität

Präzedenz: Rangfolge/Priorität der Operatoren

Assoziativität: regelt die Abarbeitung bei gleicher Priorität. Alle zweistelligen Operatoren sind links-assoziativ, Zuweisungsoperatoren rechts-assoziativ

Kontrollstrukturen

Ergebnis der Bedingung muß boolean sein!

Felder
Sind Referenztypen auf Objekte. Indizierung ab 0;
```
        <Datentyp>[] <Feldname>;
        <Feldname>=new <Datentyp>[<anzahl>];
        <Feldname>.length(); // tatsächliche Feldgröße
```
Strings
```
String <Name>; <Name> = <Zeichenkette>;
```
oder
```
<Name>=new String (<Zeichenkette>);
<Name>.length(); // String-Länge als int
<Name>.charAt(int i); // Zeichen an der Stelle i
// Position der Zeichenkette s in <Name>
<Name>.indexOf(String s);
// SubString von i1 bis i2
<Name>.substring(int i1, int i2);
```
Objekte
Klassenname immer mit Großbuchstaben beginnen!!!
```
public class <Klassennamen>
{ // Definitionen von Zustandsvariablen
  // Definitionen von Methoden }
```
Instanzen der Klasse erzeugen
```
<Klassennamen> <Objektname>;
<Objektname> = new <Klassennamen>(<Parameterliste>);
```
Zugriff auf Methoden der Instanz
```
<Objektname>.<Methode>(<Parameterliste>);
```
Alle Parameterübergaben werden Call by Value übergeben!!!
Für Call by Reference müssen eigene Objekte definiert werden. Der Zugriff auf Membervariablen geschieht über Memberfunktionen. Oder Code einfach á la Inline einbauen, z.B. bei swap().
Statische Variablen und statische Methoden
Datenelemente mit dem Vorsatz static: Existiert nur einmal im Speicher unabhängig von der Anzahl der Instanzen!
Methoden mit dem Vorsatz static: Zweck: für generelle Arbeiten einer Klasse, Zugriff nur auf Static-Variablen möglich
This
Referenz auf die eigenen Instanz. Mit this(<Parameterliste>) können von einem Konstruktor andere Konstruktoren aufgerufen werden. Jedoch muß diese Anweisung die erste im Konstruktor sein.
Beziehung zwischen Objekten
Stückliste: Relation der Form "besteht aus"
Vererbung: Relation der Form "ist ein". Eine neue Klasse entsteht durch Erweiterung einer bereits bestehenden Klasse.
In Java gibt es **keine Mehrfachvererbung!!!**
Overriding von Methoden
Die Originalmethode der Basisklassen kann durch die Referenzvariable `super.<Methode>(<Parameterliste>)` aufgerufen werden.
Schutzmechanismus
public: Zugriff von außen möglich
private: nur Klassenmethoden haben Zugriff, Variablen müssen private sein (Information hiding)
private protected: ist abgeschafft
protected: private, aber abgeleitete Klassen haben Zugriff
Ohne Schutzangabe: alle Klassen des gleichen package haben Zugriffsrechte
package: Sammlung von Klassen
```
package <Packagename>;
```
muß die erste Anweisung in einem Java Quelltext sein!!!
Konstruktoren in Klassenhierarchien
1. Aufruf des Konstuktors der Basisklasse mit `super(<Parameterliste>);`
2. Initialisierung der Klassenvariablen und Initialisierungsanweisungen
3. Ausführen des eigenen Konstruktorrumpfes
Datenvariablen in Instanzen werden beim Erzeugen der Instanz mit Defaultwerten belegt (siehe oben: Datentypen und Kontrollstrukturen).
lokale Variablen in Methoden werden nicht automatisch initialisiert.
Overloading: Methoden mit gleichen Namen, unterschiedlichen Signaturen in gleicher Klasse
Overriding: Methoden mit gleichem Namen, gleiche Signatur in Hierarchie. Java arbeitet ausschließlich mit virtuellen Funktionen.
Overriding kann auch bei Klassenvariablen genutzt werden.
Klassen/Methoden mit Vorsatz final
```
final class <Klassennamen> { ... }
```
diese Klasse kann nicht mehr als Basisklasse verwendet werden
Gründe dafür: 1. Bedeutung der Klasse kann nicht mehr redefiniert werden; 2. Compiler optimiert den Code bei final Klassen. Methoden vom final-typ können bei abgeleiteten Klassen nicht redefiniert werden.
final-Klasse: alle Methoden standardmäßig final.
Abstrakte Klassen
Hier muß mindestens eine Methode der Klasse abstrakt sein. Diese Klasse muß vererbt werden. Diese Methoden werden dann neu redefiniert. Abstrakte Klassen eignen sich immer dann, wenn generelle Konzepte zum tragen kommen. Abgeleitete Klassen übernehmen den Rest der Implementierung
Die Klasse Vector (java.util.Vector)
Variablenvereinbarung:
```
Vector <Klassennamen>;
```
Initialisierung:
```
<Klassennamen>=new Vector();
```
Elemente einfügen:
```
<Klassennamen>.addElement(object <Name>);
```
Elemente/Objekte entfernen:
```
<Klassennamen>.removeElement (object <Name>);
```
Listengröße:
```
<Klassennamen>.size();
```
Referenz auf Objekt:
```
object <Name> = <Klassennamen>.elementAt(<index von 0 bis
                                          size()-1>)
```
weitere Methoden:
```
aVector.contains( object ); // teste ob Objekt enthalten
aVector.indexOf( object );  // erstes Vorkommnis
aVector.insertElementAt( object, int );
                          // an bestimmter Pos. einfügen
aVector.removeElementAt( object, int );
                         // an bestimmter Pos. entfernen
int aVector.lastElement;    // Position letzter Eintrag
boolean aVector.isEmpty;
int aVector.capacity();     // Aufnahmevermögen des Vektors
```

Typ des Objektes mit instanceof herausfinden:
```
if   (<Name> instanceof <Klasse>);
        { <Klasse> <Klassennamen> = (<Klasse>) <Name> }
```

11. Exceptions
1. Fehlerbehandlung; 2. Code wird aufgebläht (bis zu 50% oder mehr); 3. Korrektheit steht im Gegensatz zu Klarheit; 4. stellt ProgSprache keine Methoden zur Verfügung, kommen evtl. zusätzl. Parameter an den Schnittstellen hinzu
Exceptions (Schreibweise beachten !)
1. sind Objekte, die einen Fehler- oder Ausnahmezustand anzeigen; 2. werden im Fehlerfall erzeugt und „nach außen geworfen"; 3. gehören im Prinzip zur Signatur einer Methode, müssen im Kopf mit angegeben werden; 4. geworfene Exceptions müssen im aufrufenden ProgTeil aufgefangen und bearbeitet werden (**catch**), 5. Fehlerbehandlung ist Teil der Methodenschnittstelle
11.2 Generieren eigener Exception-Typen
1. Ableiten von der Klasse Exception; 2. enthält nur Stringvariable, mit Fehlerbeschreibung werden kann; 3.bei Erzeugen des Objektes kann dieser Fehler im Konstruktor angegeben werden.
```
        Exception e = new Exception(„Fehlertext");
```
Besser: eigene Fehlerobjekte durch Vererbung von Exception zu erzeugen und mit programmabhängigen Daten zu versehen, z.B. IntMenge-Objekt: add() will bereits vorhandene Zahl einfügen, remove() eine nicht vorhandene Zahl löschen.
```
public class EnthaltenException extends Exception {
        int zahl;
        EnthaltenException (int z)
        { super („Zahl enthalten"); zahl=z; } }
```
11.3 Die throws-Klausel
Soll add() in IntMenge die Exception produzieren, muß dies im Methodenkopf von add() angegeben werden:
```
        public add(int zahl) throws EnthaltenException {
        ... }
```
Die Methode, die add() aufruft, muß in der Lage sein die Ausnahme zu behandeln (Checked Exceptions). Mehrere Exceptions einer Methode müssen nach throw mit Komma getrennt werden.
11.4 Auslösen von Exceptions
```
public void add (int zahl) throws EnthaltenException {
        ...
        if (<zahl in Menge enthalten>) throw new
            EnthaltenException(zahl); }
```
Das Werfen einer Ausnahme beendet die Methode.
11.5 Behandlung von Exceptions
1. Auffangen und Behandeln der Ausnahme (try, catch); 2. Auffangen, umwandeln in eigene Ausnahme und werfen der eigenen Ausnahme; 3. Ausnahme in der eigenen Methodendefinition bei throws aufführen, also durchreichen
```
try     {
        ...
        r.add(zahl): }
catch (EnthaltenException e)
        { // Behandeln von e }
catch (Exception X e1)
        { // Behandeln von e1 }
 ...
finally { ... }   // optional
```
Abarbeiten des try-catch-Blocks bis: 1. Exception erzeugt wurde; 2. Block fertig abgearbeitet wurde
zu 1.: Abarbeitung der catch-Klauseln von oben nach unten. Bei Hierarchien wird auch auf Superklassen geprüft. Wird eine catch-Klausel gefunden, wird der zugehörige Block ausgeführt, wenn nicht, wird die Exception an den umliegenden try-catch-Block weitergeleitet.
Ein existenter finally-Block wird in jedem Falle ausgeführt, entweder nach einem erfolgreichen try-Block oder nach einem fertig behandelten catch-Block. Außer in Fall 2. wird danach mit de Statement weitergearbeitet, welches dem gesamten Anweisungsgebilde als nächstes folgt.

12. Threads
1. in den meisten ProgSprachen wird sequentiell abgearbeitet; 2. Java erlaubt Realisierung von parallelen bzw. quasiparallelen ProgEinheiten (Threads); 3. Probleme: Deadlocks (2 Threads sperren sich gegenseitig), Synchronisation (gleichzeitiger Zugriff auf Daten); 4. Modellierungshilfe: Petri-Netze
12.2 Erzeugen von Threads
```
Thread arbeit = new Thread();
```
1. Üblicherweise werden von Klasse Thread abgeleitete Klassen erzeugt; 2. Nach Erzeugung mit new läuft der Thread noch nicht!; 3. es gibt 5 Zustände: „erzeugt", „ablauffähig", „rechnend", „wartend" und „beendet"
4. Alle Threads haben eine Priorität; Threads mit niederer Priorität laufen nur, wenn sich keinen höherprioren Prozesse (höherer Wert) im Zustand ablauffähig befinden, laufende niedere Threads werden durch höhere verdrängt (Preemptiv-System).
Threads mit gleicher Priorität werden dem Prozessor im Zeitscheibenverfahren nach Round-Robbin-Strategie zugeteilt.
```
        aThread.setPriority(int);
        actualPriority = int aThread.getPriority;
```
12.3 Beispiel für einen Thread
```
class PingPong extends Thread {
        String wort;
        int delay;
        PingPong(String was, int del)
          { wort=was; delay=del; }
        public void run(){
        try {
          for(;;) { //Endlosschleife
                    System.out.println(wort+" „);
                   sleep(delay); //Verzögerung in ms }
          }
        //nötig für sleep()
        catch (InterruptedException e) {}
}
public static void main(String[] args) {
        //Threads werden erzeugt und sofort gestartet
        new PingPong(„Ping", 53).start();
        new PingPong(„Pong", 55).start(); }
```
12.4 Synchronisation von Methoden
Vorsatz: synchronized, d.h. ausführender Thread dieser Methode sperrt dieses Objekt
Regel: ruft ein weiterer Thread eine synchronized-Methode auf, dann wird dieser Thread solange blockiert bis die Sperre für dieses Objekt aufgehoben wird (immer dann, wenn synchronized-Methode beendet wird.)
Geschachtelte Aufrufe von Sync-Methoden sind möglich, Sperre wird aber erst bei Verlassen der äußersten Methode gelöst
```
class Konto {
        private double summe;
        public Konto(double startSumme)
          { summe = startSumme; }
        public synchronized void einzahlen(double betrag)
          { summe += betrag ;}
        public synchronized void auszahlen(double betrag)
          { summe -= betrag ;} }
```
Nicht statische synchronized Methoden sperren immer das Objekt. Statische synchronized Klassenmethoden sperren die ganze Klasse, d.h. weitere static synchronized Methoden werden blockiert, aber keine Objektblockade
12.5 Synchronisierte Blöcke
Es gibt die Mögl. Teile des Codes einer Methode synchronisiert abzuarbeiten:
```
        synchronized(<objekt>) { ... }
```
In den runden Klammern muß ein Ausdruck stehen der eine Objektreferenz ergibt. Darf der ausführende Thread das Objekt betreten, wird dieses Objekt gesperrt und der im zugehörigen Block enthaltene Code ausgeführt, danach die Sperre wieder aufgehoben.
12.6 wait(), wait(long int) und notify
wait() darf nur in synchronized-Methoden aufgerufen werden. wait() löst die Objektsperre und gibt damit dieses Objekt wieder frei.
12.7 Methode yield() zur Effizienssteigerung
Thread kann durch Aufruf von yield() den Prozessor freiwillig abgeben und somit anderen Threads eine Chance zum Weiterlaufen geben. Es kann vorkommen, daß dies wieder der gleiche Thread ist, falls dieser die höchste Priorität besitzt.
12.8 Deadlocks, Sache des Programierers
Wenn 2 Threads eine Sperre auf 2 Objekte durchführen und anschließend versuchen kreuzweise auf das andere gesperrte Objekt zuzugreifen, besteht die Gefahr einer Verklemmung. Java bietet keine Möglichkeit, solche Deadlocks zu erkennen.
12.9 Eine zweite Realisierungsmöglichkeit von Threads
Problem: Keine Mehrfachvererbung in Java, d.h. vererbte Klassen könnten niemals als Thread implementiert werden.
Lösung: `class x extends Applet implements Runnable`
Ein Runnable-Objekt kann in seiner eigenen Thread-Umgebung ausgeführt werden, indem ein Thread-Objekt angelegt wird und dieses Objekt (this) in seinem Konstruktoraufruf übergeben wird. Ein so konstruiertes Thread-Objekt ruft dann beim Starten genau diese im eigentlichen Anwendungsobjekt realisierte run()-Methode auf.

13. Die awt-Library (abstract window toolkit)
Container: zum Verwalten anderer Dialogelemente, wie z.B. Buttons, Checkbox ... Frame = Fenster im Windowssystem, Panel erlaubt das Kopieren von Dialogelementen
Layout-Manager: werden innerhalb von Container-Objekten benutzt, um Dialogelemente automatisch anzupassen
Kontrollelemente: Interaktion mit dem User durch
13.2 Container und Layouts
Ein Frame hat einen äußeren Rahmen , eine Titelleiste und ist die Basisklasse in einem Java-Programm.
```
import java.awt.*;
public class FrameDemo{
        public static void main (String [] args) {
                Frame fr=new Frame(„Windows-Titel");
                fr.resize(200,200); // neue Fenstergröße
                fr.show();          // alles anzeigen
        } }
```
Borderlayout
Einzelne Elemente werden gemäß dem Kompaß angeordnet(**North, South, West, East, Center**)
```
import.java.awt.*;
public class BorderDemo {
        public static void main ( String[] args) {
                Button buttons[] = new Button[6];
                Frame fr = new Frame(„Java ist schön");
                fr.setLayout( new BorderLayout());
                fr.add(„North",button[1]);
                // korrigiert die Frame-Größe auf die
                    von den Elementen benötigte Größe
                fr.pack();
                fr.show(); } }
```
FlowLayout
Anordnung der Elemente entsprechend einer im Konstruktor angegebenen Konstanten:
```
        new FlowLayout(FlowLayout.LEFT);   // Linksbündig
        new FlowLayout(FlowLayout.RIGHT); // Rechtsbündig
```
GridLayout
```
        GridLayout(Zeilen, Spalten);
```
13.3 Menüs
```
        MenuBar mb = new MenuBar();
        Menu m = new Menu(„Menü 1");
        m.add (new MenuItem(„Erstes"));
        m.add (new MenuItem(„Zweites"));
        mb.add(m);
        fr.setMenuBar(mb);
```
13.4 Buttons, Text- und Auswahlfelder
```
        new Button(„Buttontext");
        new TextField(„defaultString");
        new Label(„Labeltext");
        Choice c = new Choice();
        c.addItem („Auswahl 1");
        c.addItem („Auswahl n");
        new Checkbox („Knopftext");
```
Danach wird das Element c mit add(c) in den Container eingefügt.
13.5 Ereignisbehandlung in Java
```
        public boolean handleEvent (Event evt)
```
Analyse in einem großen 'switch'. Aufruf der vordefinierten Methoden.
```
        public boolean action(Event evt, Object arg)
```
Dialogelement wurde manipuliert, z.B. Button gedrückt, in arg befindet sich das auszulösende Objekt.
```
        public boolean mouseDown(Event evt, int x, int y)
        public boolean mouseDrag(Event evt, int x, int y)
        public boolean mouseEnter(Event evt,int x, int y)
        public boolean mouseExit(Event evt, int x, int y)
        public boolean mouseUp(Event evt,   int x, int y)
        // Taste <key> wurde gedrückt
        public boolean keyDown(Event evt, int key)
```
Defaultfall: return-value ist false, d.h. keine Methode wird ausgeführt
Überschreiben: public **class** Button1 extends Button **{** ... **}**
Nachteile: aufwendig, jeder Button braucht eine eigene Klasse
Realisierung im übergeordneten Element:
```
public boolean action (Event ev, Object arg) {
        if(ev.target instanceof Button) {
                String str = (String) arg;
                if(str.equals("B1"))
                  { // bearbeite Button1 }
                if(str.equals("B2"))
                  { // bearbeite Button2 }
        } else if (ev.target instanceof Checkbox) { ... }
}
```
Ereignisbehandlung ab JDK 1.1
Delegation-Based-Model: Mitteilung zu GUI, über welche Ereignisse man informiert werden will. System ruft dann automatisch diese Methoden auf, wenn das Ereignis eintritt (Callback!)
Vorteile: bessere Performance, nur notwendige Ereignisse werden befördert. Observer/MVC-Pattern!
Ereignisse: werden von Ereignisquellen (Button, List, ...) erzeugt, z.B. Mausklicks, Mausbewegungen, Tastatureingaben
Ereignisempfänger: Objekte, die eine solche Ereignisschnittstelle implementieren !
Registrierung: z.B. `button.addActionListener(Empfängerobjekt);`
Ereignisse im AWT
Low-Level (unterste Ebene):
Maus(MouseListener, MouseMotionListener),
Tastatur(KeyListener),
Fenster(WindowListener),
Containeränderung(ContainerListener),
Tastaturfokus(FocusListener),
Komponentenänderung(ComponentListener)
Semantische Events:
Scrolling(AdjustmentListener),
Dialogkomponenten(ActionListener),
Auswahl (ItemListener),
Texteingabe(TextListener)

```
Button b = new Button("OK");
b.addMouseListener(  new MyMouseListener());
b.addActionListener(  new MyActionListener());  ...

class MyMouseListener implements MouseListener {
// alle MouseListener-Methoden müssen realisiert werden!!!
         public void mouseClicked(MouseEvent evt) { ... }
         public void mouseExited(MouseEvent evt) { ... }
         public void mouseEntered(MouseEvent evt) { ... }
         public void mousePressed(MouseEvent evt) { ... }
         public void mouseReleased(MouseEvent evt) { ... }
...}
class MyActionListener implements ActionListener {
         public void actionPerformed(ActionEvent evt) {
... } }
```

wichtig: alle Methoden eines Interfaces müssen realisiert werden

Besten Dank an Maggie für Ihren selbstlosen Einsatz und ihren aktiven Beitrag zur studentischen Gemeinschaft.

Event-Adapter-Klassen

Adapter implementieren die verschiedenen Ereignis-Interfaces, durch Vererbung und Überschreiben dieser Methoden.

Vorteil: Es muß nur diejenige Methode überschrieben werden, die für die Anwendung interessant ist

```
Button b = new Button("OK");
b.addMouseAdapter(  new MyMouseAdapter());

class MyMouseAdapter extends MouseAdapter {
// es muß nur die gewünschte Methode realisiert werden
         public void  mouseClicked(MouseEvent evt) { ... }
}
```

Innere Klassen

Klassen können innerhalb von anderen Klassen deklariert werden.

Vorteil: Zugriff auf Variablen und Methoden der Elternklasse. **Keine** Umkehrung mögl.

Anonyme Klassen

Haben keinen Namen. Werden direkt bei der Instanzierung deklariert. Nach new() schreibt man die Deklaration der Klasse durch Nennung der Superklasse.

```
Button b = new Button("OK");
b.addMouseListener(new MouseAdapter() {
         public void mouseClicked(MouseEvent evt) { ... }
}
```

Interfaces

Ziel: gleichartiges Verhalten von Klassen, insbesondere in Hierarchien. Nachteil: Verhalten ist vorgegeben, statisch.

Definition: Sammlung von Methodendeklarationen (abstrakt) und konstanten Werten

Verwendung: gemeinsame Schnittstellen, teilweise Realisierung von Methoden-deklaration, Anpassen der Programmierschnittstelle von Objekten, ohne die Klassenzugehörigkeit zu ändern

keine Mehrfachvererbung, weil: 1. Membervariablen können nicht vererbt werden; 2. Methodenimplementierungen können nicht vererbt werden; 3. Interface-"Hierarchie" ist unabhängig von Klassenhierarchie

```
         [public] interface Interfacename [extends
ListofSuperinterfaces]     { ... }
```

Wichtig: 1. Ein Interface kann eine Klasse erweitern; 2. die Superinterfaces werden ohne Komma getrennt; 3. ein Interface 'erbt' alle Konstanten und Methoden von seinen Superinterfaces

Rumpf: 1. Methodendeklarationen und Konstante; 2. folgende Schlüsselwörter in Variablendeklaration sind nicht erlaubt (synchronized, privat, protected); 3. alle Konstanten sind à priori: final, public, static

```
public interface Collection {
         int   MAXIMUM=300;
         void add (Object o);
         void delete (Object o);
         int  currentCount ( ); }
```

Implementierung eines Interfaces: durch Verwendung eines Schlüsselwortes **implements** welches von einer durch Komma getrennten Liste der Interfaces gefolgt wird.

Werden nicht alle geforderten Methoden realisiert, so ist die Klasse abstrakt!

```
class LIFO Stack implements Collection {
         void add (Object o) {...} }
```

Interfaces als Datentypen

1. durch ein Interface wird ein neuer Datentyp definiert; 2. gleiche Verwendung wie bei normalem Java-Datentyp; 3. der Name eines Interface kann nicht verwendt werden zur Definition einer Klasse bzw. zur Instanzierung

```
public interface Guest {
         public void checkIn ( );
         public void roomService ( );
         public void checkOut ( ); }
public interface Passenger {
         public void checkBaggage ( );
         public void board ( );
         public void; }
class Both implements Guest, Passenger { ... }
```

Wichtig: 1. ein Objekt vom Typ Both spielt verschiedene Rollen (für Hotel: Guest, für Fluggesellschaft: Passenger); 2. ein solches Objekt kann diese Rollen gleichzeitig übernehmen; 3. Objekte sind loser gekoppelt; 4.leichter in größeres System zu integrieren

```
class Airline {
         //Referenz auf Objekt, das das Interface
            passenger realisiert
         Passenger p = new Both(); }
class Hotel {
         Guest g = new Both(); }
```

Objekte vom Typ "Both" können beide Rollen übernehmen, d.h. Guest und Passenger.

Unterschiede zwischen C++ und Java

Arrays: int feld[20]; (C++), int [20] feld; (Java)

Leere: NULL (C++), null (Java)

Speicherhandling: delete erforderlich (C++), garbage collector (Java)

Postfix, Präfix: Hab das Beispiel "i++ + i++ + --i" mal ausprobiert:

Mit C ergibt das 8 und i ist erst nach dem Ende der ganzen Anweisung 4 (also wird 3 + 3 + 2 gerechnet)!

Bei Java ergibt das Ganze 11, so wie Herr Walter gesagt hat. 3 + 4 + 4. Der Grund ist der, dass Java ein interpretierendes System ist während C++ compilierend ist.

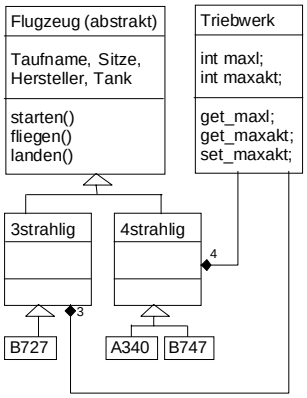
Portierung: I/O, Zeigerarithmetik, CBR, Stringmanipulation, struct, union, type

Sonstiges

1. Ein Adapter ist in diesem Sinne keine abstrakte Klasse sondern eine Klasse, bei welcher die entsprechenden Methoden leere Rümpfe besitzen (dies ist etwas anderes als abstract !!!). Damit muss man dann auch nur diejenigen Methoden realisieren, an denen man wirklich interessiert ist. Listener sind Interfaces und erfordern, dass alle Funktionen , egal ob notwendig oder nicht, realisiert werden.
2. Wie setzt man in der Regel dann CBR-Funktionsaufrufe um? Man definiert ein eigenes Objekt und dieses wird dann über die Referenzen automatisch (wegen CBV) geändert!!!!
3. x**&~**100 entspricht in C++: !(x>=0) && (x<=100)
4. Implizite Typumwandlungen werden nicht als Friend-, sondern als Memberfunktionen realisiert, z.B. `operator int()`
5. dynamische Typen haben keinen Namen
6. Bei der C++-Klassendefinition ist ein Semikolon erforderlich: class Test **{** ... **};**
7. Sonderfälle DVL: Ein-/Ausketten am Anfang, Mitte, Ende. Vorsicht bei leerer Liste.
8. Java-Strings können in Ihrer Länge nicht verändert werden. -> Stringbuffer

Vorsicht: Virtuelle Funktionen können auch default-Werte besitzen!!! Dann wird nicht in der Hierarchie gesprungen, wie es sonst immer der Fall ist.

UML



Listenverwaltung

```
class Listenverwaltung {
   Listenelement *kopf;
   public:
   Listenelement()
     { Kopf=NULL; }
   ~Listenelement()
     { for (;Kopf!=NULL;) delete kopf; }
   Listenelement *get_kopf();
   void set_kopf(Listenelement *);
   void einfuegen(Listenelemnent *); }

class Listenelement {
   char *pc;
   Listenelement *succ;
   Listenelement *pred;
   Listenverwaltung *lv;
   public:
   Listenelement(char *pc, Listenverwaltung *lv)
     { succ=pred=NULL; this->lv=lv; this->pc=new char
       [strlen(pc)+1]; this->pc=pc; lv.einfuegen(lv); }
   ~Listenelement()
   { Listenelement *hlp; delete this->pc;
     if(lv->get_kopf()==this)
       { hlp->get_succ(); hlp->set_pred(lv->get_kopf());
         lv->set_kopf(this->get_succ()); }
     else
       { hlp=this->get_pred();
         hlp->set_succ(this->get_succ());
         if (this->get_succ()!=NULL);
         { hlp=this->get_succ();
           hlp->set_pred(this->get_pred()); }
       }
   }
   Listenelement *get_succ();
   Listenelement *get_pred();
   void set_succ(Listenelement *);
   void set_pred(Listenelement *);}
```

Exceptions

```
public class ExceptionStackLeer extends Exception  {
    ExceptionStackLeer()
    { super("Stack ist leer !"); } }

public classIntegerStack  throws  ExceptionStackVoll,
ExceptionStackLeer  {
    int max =0, groesse=0;
    int[] stack;

    public IntegerStack(int groesse) {
        this.groesse=groesse;
        stack=new int[groesse];}
    public void push(int val)  {
        if(max==groesse)
            throw new ExceptionStackVoll;
        else
            stack[max++]=val;}
    public int pop() {
        if(max==0)
            throw new ExceptionStackLeer;
        else
            return stack[max--]; } }
```

AWT und Eventhandling

```
import  java.awt.*;
public class Aufgabe6 extends Frame {
    Button b1,b2;
    List ch;
    TextField tf;
    Panel p1, p2;

    Aufgabe6() { // Konstruktur
      super("Aufgabe 6 - Java sucks!");
      resize(300,300);
      setLayout(new BorderLayout());
      p1=new Panel(); p1.setLayout(new FlowLayout());
      b1=new Button("Entfernen");
      b2=new Button("Uebernehmen");
      p1.add(b1); p1.add(b2); add("South",p1);
      p2=new Panel();
       p2.setLayout(new FlowLayout(FlowLayout.LEFT));
      ch=new List(10,false); ch.addItem("Erstes");
      p2.add(ch); tf=new TextField(10); p2.add(tf);
      add("Center",p2); show(); }

      public boolean action(Event ev, Object arg) {
        if (ev.target instanceof Button) {
          String str=(String)arg;
          if (str.equals("Entfernen")) {
              str=ch.getSelectedItem();
              int index=ch.getSelectedIndex();
              tf.setText(str); ch.delItem(index); }
           else if (str.equals("Uebernehmen")) {
               str=tf.getText(); ch.addItem(str);
               return true; }
          } return false;}
       public static void main(String [] args) {
         Aufgabe6 a=new Aufgabe6();}
}
```

Einfaches Beispiel für Eventlistener:

```
public class MyButton extends Applet implements
ActionListener {
   Button b;
   public init() {
      b = new Button(„b");
       b.addActionListener(this);
       // oder b.addActionListener(new myActionListener) }
    public void actionPerformed(ActionEvent e) {
      // Aktion.... }
}
```

Alternative

```
public class myActionListener implements ActionListener {
     public void actionPerformed(ActionEvent e) {
       // Aktion.... }
}
```

Beispiel für's AWT

```
import java.awt.*;
public class CheckAction extends java.applet.Applet {
     CheckboxGroup myChkboxGrp  = new CheckboxGroup();
     public void init() {
        add(new Checkbox(„Blau", myChkboxGrp , false));
        add(new Checkbox(„Rot", myChkboxGrp , false));
        add(new Checkbox(„Grün", myChkboxGrp , false));
        add(new Checkbox(„Gelb", myChkboxGrp , true));
        setBackground(Color.yellow); }
     public boolean action(Event evt, Object welcheAktion){
     // Überprüfung, ob ein Kontrollkästchen oder ein
        Radiobutton, falls nicht Rückgabewert false
        if (!(evt.target instanceof Checkbox)) {
          return false; }
     // Instanz des Ereignisses
        Checkbox welcheAuswahl = (Checkbox)evt.target;
        boolean checkboxStatus = welcheAuswahl.getState();
        if (welcheAuswahl.getLabel() == „Blau") {
          if (checkboxStatus) {
            setBackground(Color.blue); }
         return true; }
        if (welcheAuswahl.getLabel() == „Rot") {
          if (checkboxStatus) {
            setBackground(Color.red); }
         return true; }
       return false; }
}
```

Komplexeres Eventhandling Beispiel (java swing 1.1):

```
public class MouseEventDemo ... implements MouseListener {
                                ...
     //where initialization occurs:
     //Register for mouse events on blankArea and applet
     blankArea.addMouseListener(this);
     addMouseListener(this); }

     public void mousePressed(MouseEvent e) {
       saySomething(„Mouse pressed; # of clicks:
                    „+ e.getClickCount(), e); }
     public void mouseReleased(MouseEvent e) {
       saySomething(„Mouse released; # of clicks:
                    „+ e.getClickCount(), e); }
     public void mouseEntered(MouseEvent e) {
       saySomething(„Mouse entered", e); }
     public void mouseExited(MouseEvent e) {
       saySomething(„Mouse exited", e); }
     public void mouseClicked(MouseEvent e) {
       saySomething(„Mouse clicked (# of clicks:
                    „+ e.getClickCount() + „)", e); }
     void saySomething(String eventDescription,
                       MouseEvent e) {
       textArea.append(eventDescription + „detected on"
           + e.getComponent().getClass().getName()
           + „." + newline); }
    }
```

Stack mit Vector-Klasse

Vorsicht: Vector-Klasse kann keine Basistypen aufnehmen, z.B. int, float, byte.

```
import java.util.Vector;
public class Stack throws ExceptionStackVoll,
ExceptionStackLeer {
    Vector Stack;
    int maxSize = 0;
    Stack(int) { Stack = new Vector(size);
       maxSize = size; }
    void push(int val) {
        if(Stack.lastElement == maxSize) {
          throw new ExceptionStackVoll();}
        else {
          Stack.insertElementAt(val,  Stack.lastElement+1);}
    }
    int pop() {
        if(Stack.isEmpty()) {
          throw new ExceptionStackLeer(); }
        else {
         return (int) Stack.removeElementAt(
                                  Stack.lastElement(); }
    }
}

Alle aufgelisteten Programme ohne Gewähr!!!
```