3 Sprachkonzepte II: Einführung in die objektorientierte Programmierung

Inhalt des Kapitels im Überblick:

- Klassen und Objekte
- Referenz- und Wertsemantik
- Garbage Collector
- Klassenvariablen und -methoden
- Vererbung und Mehrfachvererbung
- Polymorphie
- Dynamisches Binden
- Überladene Methoden
- Abstrakte Klassen
- Interfaces
- Umgang mit Klassenbibliotheken (in der Übung)
  - Grafische Benutzeroberflächen
  - Datei-Ein/Ausgabe

Gliederung des Kapitels:

3 Sprachkonzepte II: Einführung in die objektorientierte Programmierung 1
3.1 Motivation 3
3.2 Klassen und Objekte 3
3.2.1 Klassendefinition 3
3.2.2 Anlegen von Objekten (Instanzen) 4
3.2.3 Zugriff auf Attribute und Methoden 5
3.3 Klassenvariablen/-methoden und Instanzvariablen 6
3.4 Referenz- und Wertsemantik 7
3.4.1 Vergleich von Objekten bei Referenzsemantik 8
3.4.2 Dereferenzierung und Garbage Collector 9
3.5 Vererbung 9
3.5.1 Polymorphie und späte Bindung 11
3.5.2 Überladene Methoden 12
3.5.3 Abstrakte Klassen und Methoden 13
3.5.4 Mehrfachvererbung vs. Interfaces 14
3.6 Modularisierung: Packages 17
3.7 Modifikatoren in Java (Zusammenfassung) 18
3.7.1 Modifikatoren für Java-Klassen 18
3.7.2 Modifikatoren für Variablen 19
3.7.3 Modifikatoren für Methoden 19

3.1 Motivation

Objektorientiertes Programmieren ist ein Programmierparadigma. Andere Beispiele für Programmierparadigmen sind imperative (Pascal, C), funktionale (Lisp, Haskell) und deklarative (Prolog) Programmierung.

Was möchte man durch Objektorientierung erreichen?

- Verkürzung der Entwicklungszeit
- Senkung der Fehlerrate
- verbesserte Erweiterbarkeit und Anpassungsfähigkeit

Maxime der objektorientierten Programmierung ist die Entwicklung allgemein wieder verwendbarer und anpassbarer Softwarebibliotheken.

Hauptmerkmale:

- Datenkapselung:
  - genau definierte Schnittstellen
  - Verbergen der Implementierungsdetails
- Vererbung:
  - einfache Modifikation und Erweiterung von bereits vorhandenen Komponenten
- Polymorphie:
  - gleiche Funktionalität für verschiedene Datentypen
  - Datentypabhängige Semantik von Operatoren und Funktionen

3.2 Klassen und Objekte

Man unterscheidet Klassen und die eigentlichen Objekte (Instanzen).

3.2.1 Klassendefinition

Eine **Klassendefinition beschreibt** die **Attribute** (attributes, Instanzvariable) **und Methoden** (methods, Prozeduren, Funktionen) eines Objektes.

Spezielle Methoden innerhalb der Klassendefinition:

Der **Konstuktor** (constructor) ist legt die Anfangswerte der Atrribute fest und führt ggf. Methoden zur Initialisierung aus.

```
Klasse → Anlegen des Objektes (Aufruf des Kostruktors) → Objekt (Instanz)
```

Der **Destruktor** (destructor, finalizer) entfernt ein dynamisch erzeugtes Objekt aus dem Hauptspeicher und führt ggf. vorher Aufräumarbeiten aus.

Beachte: Durch die Definition einer Klasse werden noch keine Objekte erzeugt. Dies erfolgt erst bei der Deklaration von Variablen oder durch dynamisches Erzeugen eines Objektes mit der `new`-Anweisung.

Syntax (Java, vereinfacht):

```
ClassDeclaration: [Modifiers] class Identifier
                        [ extends Type ]
                        [ implements Type {, Type } ]
                        ClassBody
ClassBody: { { VariableDeclaration }
             { ConstructorDeclarartion }
             { MethodDeclaration }
           }
VariableDeclaration: [Modifiers] Declaration
ConstructorDeclaration: Identifier ( [FormalArguments] ) Block
Modifiers: <siehe Text>
```

Mit dem Schlüsselwort `class` wird die Deklaration ein Klasse eingeleitet. Auf die Bedeutung von `extends`, `implements` und die möglichen `Modifiers` wird später eingegangen.

3.2.2 Anlegen von Objekten (Instanzen)

Ein **Objekt** ist eine Einheit von Daten (data) und Funktionen (functions), die auf den Daten operieren. Die Struktur von Daten und Funktionen gleichartiger Objekte sind in ihrer gemeinsamen Klasse (class) definiert.

Syntax (Java, vereinfacht):

```
StatementExpression: ... |
                     MethodInvocation |
                     CreationExpression
CreationExpression: new Identifier ( [ActualArguments] )
```

Bemerkungen:

- Mit dem Schlüsselwort `new` wird ein Objekt dynamisch erzeugt.
- Beim Anlegen des Objektes wird eine Instanz der Klasse erzeugt. Man bezeichnet ein Objekt deshalb auch als **Instanz**. Beim Anlegen wird der Konstruktor aufgerufen.
- Die Variablen eines Objektes sind die Attribute.
- Die Funktionen eines Objektes sind die Methoden. Sie definieren das Verhalten des Objektes.
- Jedes Objekt besitzt eine eigene Identität (identity) und einen eigenen Satz Daten.

3.2.3 Zugriff auf Attribute und Methoden

Innerhalb des Objektes werden die Attribute und Methoden mit ihrem einfachen Namen angesprochen. Attribute und Methoden fremder Objekte werden über ihren qualifizierten Namen (**qualified name**) `Objektname.Variablenname` bzw. `Objektname.Methodenname()` angesprochen.

Anmerkung: Es ist auch möglich, Attribute und Methoden innerhalb des Objektes mit `this.Variablenname` bzw. `this.Methodenname()` anzusprechen.

Syntax (Java, vereinfacht):

```
Primary: ... |
        FieldAccess
FieldAccess: Primary . Identifier
```

Bemerkungen:

- Primary muss Verweistyp haben, d.h. eine Referenz sein und zugehörige Klasse muss Attribut bzw. Methode mit angegebenem Namen haben
- Laufzeitfehler, wenn Auswertung des Primary `null` liefert (`Null pointer exception`)

Beispiel:

Klassendefinition:

```
class Bruch {

    /** Es folgen die Attribute */
    int zaehler;
    int nenner;

    /** Konstruktor */
    Bruch (int z, int n) {
        zaehler = z;
        nenner = n;
        kuerze();
    }

    /** Hier eine Methodendefinition zum Kürzen */
    void kuerze () {
        ...
    }

    /** Methode zum Hinzuaddieren */
    void add(Bruch r) {
        zaehler = zaehler*r.nenner + r.zaehler*nenner;
        nenner = nenner*r.nenner;
        kuerze();
    }

    /** Methode zum Wandeln in einen String */
    public String toString() {
        return "(" + zaehler + "/" + nenner + ")";
    }

}
```

Anlegen eines Objektes:

```
class BruchApplication {

    static void main (String[] argv) {

        Bruch b = new Bruch (2,6);    // Anlegen des Bruchs 2/6 mit new
        b.add(new Bruch(1,2));        // Hinzuaddieren von 1/2
        System.out.println("b=" + b); // ruft implizit b.toString() auf
                                      // ... rauskommen sollte 5/6
    }
}
```

3.3 Klassenvariablen/-methoden und Instanzvariablen

Klassenvariablen sind (im Gegensatz zu Instanzenvariablen) für alle Objekte einer Klasse nur einmal vorhanden. **Klassenmethoden** sind Prozeduren oder Funktionen einer Klasse, die ohne Bezug zu einem Objekt aufgerufen werden können.

Klassenmethoden können nicht auf Instanzvariablen, sondern nur auf Klassenvariablen operieren. Sie werden in Java mit dem Schlüsselwort (Modifier) `static` deklariert.

Beispiel:

```
class TestKlasse {
    static int laufendeNummer = 0; // ... ist eine Klassenvariable
    int eigeneNummer;
    TestKlasse() {
        laufendeNummer++;
        eigeneNummer=laufendeNummer;
    }
}

class TestApplication {
static void main (String[] argv) {  // ... ist eine Klassenmethode
        TestKlasse t1 = new TestKlasse();
        TestKlasse t2 = new TestKlasse();
        TestKlasse t3 = new TestKlasse();
        System.out.println(t1.eigeneNummer);
        System.out.println(t2.eigeneNummer);
        System.out.println(t3.eigeneNummer);
    }
}
```

28.6.2001

Jede Instanz von `TestKlasse` hat eine eigene Identität, die in der Variable `eigeneNummer` gespeichert wird. Anwendung z.B. bei `Threads`.

Die Klasse `TestApplication` besitzt nur eine statische Methode, d.h. eine Klassenmethode und muss deshalb nicht instantiiert werden.

Beachte: Klassenmethoden können nicht auf Instanzvariablen operieren:

```
class AnotherTestApplication {
    int instanzvariable = 5;
    static void main (String[] argv) {  // ... ist eine Klassenmethode
        System.out.println("instanzvariable = " + instanzvariable);
    }
}
```

Liefert Compilerfehler:

```
javac AnotherTestApplication.java
AnotherTestApplication.java:4: Can't make a static reference to nonstatic
variable instanzvariable in class AnotherTestApplication.
        System.out.println("instanzvariable = " + instanzvariable);
                                                  ^
1 error
```

Aber das klappt: (rein statisch)

```
class AnotherTestApplication {
    static int instanzvariable = 5;     // ist KEINE Instanzvariable
    static void main (String[] argv) {  // ist eine Klassenmethode
        System.out.println("instanzvariable = " + instanzvariable);
    }
}
```

Oder so: (objektorientiert)

```
class AnotherTestApplication {
    int instanzvariable = 5;            // ist eine Instanzvariable (ohne
static)
    static void main (String[] argv) {  // ist eine Klassenmethode
        AnotherTestApplication app = new AnotherTestApplication();
        System.out.println("instanzvariable = " + app.instanzvariable);
    }
}
```

3.4 Referenz- und Wertsemantik

Klasse: „Schablone“, vergleichbar mit der Vereinbarung eines Produkttyps

Objekt (Instanz): „Behälter“ für Werte dieses Typs, Variablen sind nicht statisch vereinbart, sondern werden dynamisch erzeugt

→ In der Objektvariable wird dann eine **Referenz** auf das Objekt gespeichert.

Beispiel:

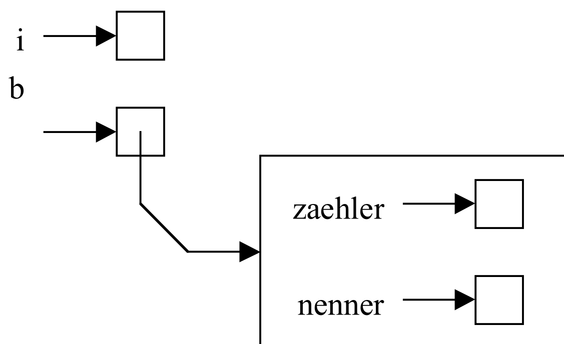
Deklaration:

```
int i;
Bruch b;              // Deklaration einer Variable vom Typ Bruch
```

Initialisierung:

```
c = new Bruch(1,4);   // Initialisierung
```

Situation nach Initialisierung:



Variable `b` vom Typ `Bruch` enthält Verweis (Referenz, reference, pointer) auf Objekt der Klasse `Bruch`. `Bruch` ist ein Verweis- oder Klassentyp.

Beispiel: Vergleich Wert- und Referenzsemantik

```
Bruch r1 = new Bruch(3,4);
Bruch r2 = new Bruch(1,8);
r1.add(r2); // r1 = 7/8
r2 = r1;
r1.add(r1); // r1 = 7/4
            // r2 = ?
```

Bei Wertsemantik bleibt `r2` unverändert `7/8`, bei Referenzsemantik zeigen `r1` und `r2` am auf das gleiche Objekt, deshalb ist `r2 = 7/4`.

Generell unterscheidet man in objektorientierten Sprachen:

Wertsemantik: Bei der Zuweisung wird der Inhalt eines Objektes in ein anderes Objekt kopiert. Nach der Zuweisung gibt es zwei verschiedene Objekte mit identischem Zustand.

Referenzsemantik: Bei der Zuweisung wird lediglich eine Objektreferenz (Speicheradresse) kopiert.

Folge einer Referenzsemantik:

- Mehrere Variable können sich auf das gleiche Objekt beziehen.
- Das alte referenzierte Objekt kann nicht mehr angesprochen werden und deshalb automatisch aufgeräumt werden (→ Garbage Collector)

Java arbeitet mit Referenzsemantik!

3.4.1 Vergleich von Objekten bei Referenzsemantik

Die Vergleichsoperatoren `==` und `!=` vergleichen nur die Referenzen zweier Objekte!

Beispiel:

```
Bruch b1 = new Bruch(1,2);
Bruch b2 = new Bruch(1,2);
if (b1 == b2)       System.out.println("Referenz gleich");
else                System.out.println("Referenz ungleich")
if (b1.equals(b2))  System.out.println("Inhalt gleich");
else                System.out.println("Inhalt ungleich")
```

Ausgabe:

```
Referenz ungleich
Inhalt gleich
```

Voraussetzung: Implementation von `equals()` *in Klasse* `Bruch`:

```
public boolean equals(Bruch b) {
    return (b.zaehler == zaehler) && (b.nenner == nenner);
}
```

Analoges gilt für **Vergleiche in anderen Klassen z.B.** `String`:

```
String s1 = "ALP2 macht Spass!";
String s2 = "ALP2 MACHT SPASS!";
boolean r;
r = s1.equals(s2); // → true, falls Inhalte von s1 und s2 gleich; hier: false
r = s1.equalsIgnoreCase(s2); // ignoriert Gross-/Kleinschreibung; hier: true
int c = s1.compareTo(s2);    // lexikographischer Vergleich
                             // c < 0 wenn s1 <  s2
                             // c > 0 wenn s1 >  s2 (hier)
                             // c = 0 wenn s1 == s2
```

3.4.2 Dereferenzierung und Garbage Collector

Wass passiert mit Objekten, für die keine Referenz mehr exisitiert? → Sie müssen aufgeräumt werden…

`null` ist der Initialisierungswert von Instanzvariablen. Instanzvariablen, die den Wert `null` haben bedeuten: Die Referenz verweist auf „nichts“.

Der Speicherplatz von Objekten die nicht mehr referenziert sind, wird in Java durch den **Garbage Collector** (GC) freigegeben.

- automatische Speicherverwaltung
- erkennt, welche Objekte nicht mehr ansprechbar sind
- GC ist ein „leichtgewichtiger Hintergrundprozess“ (Thread) — kostet damit Performance

Alternative: (z.B. C++)

- **Explizite Freigabe** durch Aufruf des Destruktors.
- Vermeidet Laufzeitnachteile, verursacht in der Praxis aber häufig schwerwiegende Programmierfehler

Beachte: In Java kann in jeder Klasse eine `finalize()`-Methode definiert werden, die automatisch vom GC aufgerufen wird, um eigene „Aufräumarbeiten“ durchzuführen.

Beispiel: Erweiterung von Klasse `Bruch`

```
public void finalize() {
      System.out.println("finalize() Bruch "+this.toString());
      zaehler=0; nenner=0; // Initialisierung
}
```

Datensicherheit: Speicherplätze werden explizit gelöscht, um anderen Anwendungen nicht unberechtigten Zugriff auf zufällig vorhandene sensitive Inhalte im freigegebenen Speicher zu geben.

5.7.2001

3.5 Vererbung

Innerhalb einer Vererbungshierarchie beschreibt eine Oberklasse (Basisklasse) die allgemeineren, umfassenden Aspekte, während die abgeleitete Klasse (Unterklasse) spezielle zusätzliche Eigenschaften ausdrückt.

Eine abgeleitete Klasse

- erbt die Attribute und Methoden der Oberklasse
- kann zusätzliche Attribute und Methoden definieren
- kann ggf. das Verhalten der ererbten Methoden modifizieren

Vererbung wird durch das Schlüsselwort `extends` realisiert: (siehe auch Syntax in Abschnitt 3.2.1)

```
class Unterklasse extends Oberklasse { ... }
```

Mit `super` wird (wenn nötig) der Konstruktor bzw. die Methode der Oberklasse aufgerufen.

Beispiel: Klasse Figur und Klasse Kreis

```
class Figur {
	protected float xPos;
	protected float yPos;
	public Figur(float x, float y) {
		xPos=x;
		yPos=y;
	}
	public String toString() {
		return "Figur("+xPos+","+yPos+")";
	}
}
class Kreis extends Figur {
	protected float radius;
	public Kreis(float x, float y, float r) {
		super(x,y); // Aufruf des Konstruktors der Oberklasse
		radius=r;
	}
	public String toString() { // wird überladen
//		return "Kreis("+xPos+","+yPos+","+radius+")";
		return "Kreis("+xPos+","+yPos+","+radius+") is a
"+super.toString();
	}
}
```

Benutzung:

```
Figur f = new Figur(1,1);
System.out.println(f);
Kreis k = new Kreis(1,2,3);
System.out.println(k);
```

Ausgabe:

```
Figur(1.0,1.0)
Kreis(1.0,2.0,3.0) is a Figur(1.0,2.0)
```

Klasse `Kreis` verfügt zusätzlich über das Attribut `radius`. Die Methode `toString()` wurde *überschrieben*. Mit `super.toString()` wird die Methode `toString()` der Oberklasse aufgerufen.

Beispiel: Threads führen parallele Funktion aus. Hochzählen eines Zählers entsprechend dem übergebenen Inkrement.

```
class ThreadDemo extends Thread { // ThreadDemo ist eine Unterklasse von
Thread

	private int inkrement;
	private int zaehler=0;
	private int eigeneNummer;
	private static int laufendeNummer=0;

	public ThreadDemo(int i) {
		inkrement = i;
		laufendeNummer++;
		eigeneNummer=laufendeNummer;
	}

	public void run() {
		while(true) {
			System.out.println("Thread No. "+eigeneNummer+
				": zaehler="+zaehler);
			zaehler += inkrement;
		}
	}

	static void main (String[] argv) {
		ThreadDemo t1 = new ThreadDemo(1);
		ThreadDemo t2 = new ThreadDemo(2);
		ThreadDemo t3 = new ThreadDemo(3);
		t1.start();
		t2.start();
		t3.start();
		System.out.println("Ende main()");
	}
}
```

Ausgabe: (z.B.)

```
Ende main()
Thread No. 1: zaehler=0
Thread No. 2: zaehler=0
Thread No. 3: zaehler=0
Thread No. 1: zaehler=1
Thread No. 2: zaehler=2
Thread No. 3: zaehler=3
Thread No. 1: zaehler=2
Thread No. 2: zaehler=4
Thread No. 3: zaehler=6
...
```

Ablauf: `start()` initiiert das Ablaufen des Threads. Die *überschriebene* `run()`-Methode wird durch `start()` aufgerufen. Durch die Java-Umgebung bzw. das Betriebssystem bekommt jeder Thread Rechenzeit zugeteilt → quasi-parallele Abarbeitung der Threads.

3.5.1 Polymorphie und späte Bindung

Es ist erlaubt, einer Variablen ein Objekt zuzuweisen, das vom gleichen Typ wie die Variable ist oder einem davon abgeleiteten Typ.

Polymorphie bezeichnet die Eigenschaft, dass eine Variable Objekte unterschiedlichen Typs speichern kann.

Beispiel:

```
Figur[] ff = new Figur[3];
ff[0] = new Figur(1,1);
ff[1] = new Kreis(1,2,3);
ff[2] = new Figur(1,3);
for(int i=0;i<3;i++)
      System.out.println(ff[i]);
```

Ausgabe:

```
Figur(1.0,1.0)
Kreis(1.0,2.0,3.0)
Figur(1.0,3.0)
```

Beobachtung:

- Für `ff[1]` (obwohl vom Typ `Figur`) wird `toString()` aus der abgeleiteten Klasse (`Kreis`) aufgerufen. (wie man das erwartet…)
- Aufruf der Methode richtet sich nicht nach dem *statischen Typ*, mit dem das Array definiert wurde, sondern nach dem *dynamischen Typ*, den das Objekt besitzt.
- Verschiedene Unterklassen ein und derselben Oberklasse können unterschiedlich auf den gleichen Methodenaufruf reagieren.
- Entscheidung über aufzurufende Methode fällt erst zur Laufzeit

  → Man spricht von später Bindung.

9.7.2001

Begriffe:

- **Statischer Typ:** in der Variablendeklaration festgelegter Datentyp
- **Dynamischer Typ:** beim Erzeugen des Objektes festgelegter Typ

Frühe Bindung	Späte Bindung
• Auswahl der Methode richtet sich nach statischem Typ • Bindung wird vom Compiler vorgenommen	• Auswahl der Methode richtet sich nach dynamischem Typ • Bindung kann erst zur Laufzeit vorgenommen werden

Späte Bindung bringt Laufzeitnachteile mit sich. Manche Sprachen fordern die explizite Kennzeichnung spät zu bindender Methoden (virtuelle Methoden).

3.5.2 Überladene Methoden

Zweck: Verbessern der Universalität einer Klasse

Man sagt, eine „**Methode ist überladen**“, wenn es innerhalb einer Klasse mehrere Methoden gleichen Namens gibt, die sich durch ihre formalen Argumente unterscheiden.

Beispiel:

> `System.out.println(String s)` gibt die Zeichenkette `s` gefolgt von einem Zeilenvorschub auf der Konsole aus.
>
> `System.out.println(float f)` gibt die Fließkommazahl `f` gefolgt von einen Zeilenvorschub aus.
>
> `System.out.println()` gibt nur einen Zeilenvorschub aus.
>
> `println` ist definiert in der Klasse `PrintStream` im Package `java.io` und in insgesamt in 10 Varianten vorhanden.

Vorteil: Man muss sich nur `println` merken, nicht für jede Methode einen eigenen Namen.

Beispiel: Die Methode `add` der Klasse `Bruch` wird überladen: Addieren einer Integer-Zahl

```
/** Methode zum Hinzuaddieren (eines Bruchs) */
void add(Bruch r) {
    zaehler = zaehler*r.nenner + r.zaehler*nenner;
    nenner = nenner*r.nenner;
    kuerze();
}
/** Methode zum Hinzuaddieren einer Integer-Zahl */
void add(int x) {
    zaehler += x*nenner;
    kuerze();
}
```

3.5.3 Abstrakte Klassen und Methoden

Zweck: Manchmal möchte man eine allgemeine Klasse (oder auch nur Methode) definieren, die nicht direkt instantiiert bzw. aufgerufen werden kann, sondern erst nach der Vererbung, weil deren Verhalten lediglich abstrakt beschrieben wurde, das heißt noch nicht konkret implementiert ist.

Beispiel: Abstrakte Klasse `Fortbewegungsmittel` besitzt (abstrakte) Methode `anhalten()`.

- Unterklasse `Strassenfahrzeug` implementiert anhalten() z.B. als „Bremsbacken gegen Bremsscheibe pressen", während die
- Unterklasse `Duesenflugzeug` die Methode `anhalten()` u.a. durch „Triebwerke auf Gegenschub" implementiert.

> **Abstrakte Klassen** sind Klassen, die nur geerbt, aber nicht direkt instantiiert werden können. Von **abstrakten Methoden** sind nur die Methodenköpfe definiert. Der Body muss von der Unterklasse implementiert werden.

Abstrakte Klassen und Methoden werden durch den Modifier `abstract` realisiert: (siehe auch Syntax in Abschnitt 3.2.1)

```
abstract class Unterklasse extends Oberklasse { ... }

abstract ResultType Methodenname ( [FormalArguments] ) ;
```

Wenn eine abstrakte Methode definiert wird, muss die gesamte Klasse abstrakt definiert werden.

Beispiel: Bei `Figur` handelt es sich in Wirklichkeit nur um eine Koordinate und nicht um eine `Figur`. Definieren als abstrakte Klasse:

```
            abstract class Figur {
                ...
                abstract float flaecheninhalt() ;
                ...
            }
            class Kreis extends Figur {
                protected float radius;
                public float flaecheninhalt() {
                    ...
                }
                ...
            }
            class Quadrat extends Figur {
                protected float x_length;
                public float flaecheninhalt() {
                    return x_length*x_length;
                }
                ...
            }
            class Rechteck extends Quadrat {
                protected float y_length;
                public float flaecheninhalt() {
                    return x_length*y_length;
                }
                ...
            }
            ...
//Fehler:   Figur f = new Figur(1,1); // Figur ist abstrakte Klasse!
            Kreis k = new Kreis(1,1,2);
```

Die Methode `flaecheninhalt()` in `Figur` wurde als abstrakte Methode definiert. → Alle erbenden Klassen müssen `flaecheninhalt()` imlementieren.

Es hängt von der Problemmodellierung ab, ob und wann eine abstrakte Klasse sinnvoll ist.

3.5.4 Mehrfachvererbung vs. Interfaces

Problem: Eine Klasse möchte (unterschiedliche) Funktionen von zwei oder mehreren Oberklassen verwenden.

Beispiel:

1. Die Thread-Anwendung aus Abschnitt 3.5 soll in einem Fenster realisiert werden. Jede Nummer soll in einem Label erscheinen.
2. Eine Fensterimplementation (frame) soll ebenfalls ein Listener für Events (Mausklicks, Tastatureingaben etc.) werden.

3.5.4.1 Mehrfachvererbung

Naheliegende Idee: **Mehrfachvererbung (multiple inheritance)**

- Unterklasse erbt von mehreren Oberklassen
- → Funktionen beider Oberklassen stehen der Unterklasse zur Verfügung

Nicht funktionierendes **Beispiel**: Das Thread-Beispiel (Zähler) aus Abschnitt 3.5 soll in drei Fenstern laufen:

```
class ThreadDemo3Frames extends Frame, Thread { // das wird nicht
funktionieren

    private int inkrement;
    private int zaehler=0;
    private TextField tf = new TextField();

    public ThreadDemo3Frames(int i) {
        super("Frame No. "+i);  // Konstruktor der Oberklasse aufrufen
        this.add(tf);           // Textfeld dem Frame hinzufuegen
        this.setVisible(true);  // Frame sichtbar machen
        inkrement = i;
    }

    public void run() {
        while(true) {
            tf.setText(""+zaehler); // Zaehlerstand in Textfeld anzeigen
            zaehler += inkrement;
        }
    }

    static void main (String[] argv) {
        // Frame-Objekte erzeugen
        ThreadDemo3Frames f1 = new ThreadDemo3Frames(1);
        ThreadDemo3Frames f2 = new ThreadDemo3Frames(2);
        ThreadDemo3Frames f3 = new ThreadDemo3Frames(3);
        // Threads starten
        f1.start();
        f2.start();
        f3.start();
    }
}
```

Warum wurde in Java auf Mehrfachvererbung verzichtet?

Potentielles Problem: Zwei Oberklassen definieren Methoden mit gleichem Kopf, aber unterschiedlicher Funktionalität. Wie werden Namenskonflikte ausgeräumt?

Ausweg: Anstelle von Mehrfachvererbung werden sogenannte **Interfaces** von der Sprache unterstützt:

Andere Sprachen (z.B. C++) unterstützen Mehrfachvererbung.

12.7.2001

3.5.4.2 Interfaces

Ein Interface ist eine Klasse mit ausschließlich abstrakten Methoden und konstanten Attributen.

In Java sind alle Elemente eines Interfaces *implizit* mit den Modifikatoren `abstract` für Methoden und `final static` für Attribute versehen.

Definition eines Interface:

Syntax (Java, vereinfacht):

```
InterfaceDeclaration: interface Identifier [ extends Type {, Type } ]
```

Ein Interface kann von mehreren Interfaces abgeleitet werden!

Benutzung des Interface:

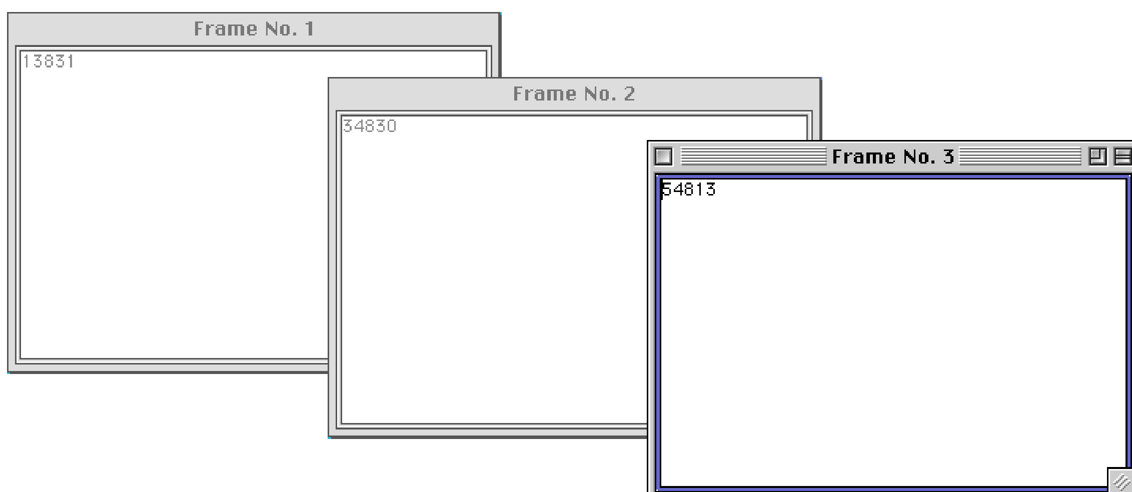
Es wird mit dem Schlüsselwort `implements` angezeigt, welche Interfaces zu implementieren sind: (siehe auch Syntax in Abschnitt 3.2.1)

```
class Unterklasse [ extends Oberklasse ] implements Interface { ... }
```

Beispiel: Da es häufig vorkommt, dass man eine Klasse als `Thread` ausführen möchte, aber von einer anderen Oberklasse erben muss, existiert für `Thread` das Interface `Runnable`.

```
import java.awt.*;
class ThreadDemo3Frames extends Frame implements Runnable {

    private int inkrement;
    private int zaehler=0;
    private TextField tf = new TextField();

    public ThreadDemo3Frames(int i) {
        super("Frame No. "+i);  // Konstruktor der Oberklasse aufrufen
        this.add(tf);           // Textfeld dem Frame hinzufuegen
        this.setVisible(true);  // Frame sichtbar machen
        inkrement = i;
    }

    public void run() {
        while(true) {
            tf.setText(""+zaehler); // Zaehlerstand in Textfeld anzeigen
            zaehler += inkrement;
        }
    }

    static void main (String[] argv) {
        // Frame-Objekte erzeugen
        ThreadDemo3Frames f1 = new ThreadDemo3Frames(1);
        ThreadDemo3Frames f2 = new ThreadDemo3Frames(2);
        ThreadDemo3Frames f3 = new ThreadDemo3Frames(3);
        // Thread-Objekte erzeugen
        Thread t1 = new Thread(f1);
        Thread t2 = new Thread(f2);
        Thread t3 = new Thread(f3);
        // Threads starten
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Ausgabe: Die Zähler aller drei Frames verändern sich.



Sehr häufige Verwendung von Interfaces: Event-Handling bei graphischen Benutzeroberflächen.

3.6 Modularisierung: Packages

Logisch zusammen gehörende Klassen sollten in Paketen (packages) zusammengefasst werden.

Beispiel:

Die Packages der Java-Standardklassenbibliothek

Syntax (Java, vereinfacht):

```
CompilationUnit: package Identifier ;
ImportDeclaration: import Identifier { . Identifier } [ .* ] ;
```

Semantik:

- Auf logischer Ebene wird eine Klasse mit Hilfe des Schlüsselworts `package` zugewiesen.
- Auf physischer Ebene werden Klassen eines Pakets in Unterverzeichnissen abgespeichert, die den Paketnamen tragen.
- Verwendung von Klassen aus Packages:
  - entweder importieren (alle Klassen des Packages mit `*` oder einzelne Klassen mit deren Namen)
  - oder unmittelbar qualifizieren beim Benutzen der Klasse (siehe Beispiel).

Beispiel:

Qualifizieren beim Benutzen

```
java.awt.Frame myframe = new java.awt.Frame("Frame Test");
myframe.add(new java.awt.Label("Das ist ein Label",java.awt.Label.CENTER));
myframe.show();
```

Importieren jeder einzelnen Klasse

```
import java.awt.Frame;
import java.awt.Label;
Frame myframe = new Frame("Frame Test");
myframe.add(new Label("Das ist ein Label",Label.CENTER));
myframe.show();
```

Importieren aller Klassen des Packages `java.awt`:

```
import java.awt.*;
Frame myframe = new Frame("Frame Test");
myframe.add(new Label("Das ist ein Label",Label.CENTER));
myframe.show();
```

Ergebnis: (immer gleich)

Anmerkungen:

- Je genereller Klassen importiert werden, umso länger benötigt der Compiler beim übersetzen.
- Bei Namenskonflikten hilft nur das Qualifizieren bei der Benutzung.

3.7 Modifikatoren in Java (Zusammenfassung)

Es sollen nachfolgend noch einmal alle Modifikatoren für Java-Klassen, Variablen und Methoden zusammengefasst werden.

3.7.1 Modifikatoren für Java-Klassen

Modifikator	Bedeutung
keiner	Klasse ist nur aus demselben Package erreichbar.
`abstract`	Klasse ist abstrakt, d.h. kann nur geerbt, aber nicht instantiiert werden.
`public`	Klasse ist von überall erreichbar.
`final`	Klasse ist final, d.h. kann nicht vererbt werden. Es können keine Unterklassen abgeleitet werden.

3.7.2 Modifikatoren für Variablen

Modifikator	Bedeutung
keiner	Variable ist nur innerhalb der eigenen Klasse und des Package erreichbar, zu dem sie gehört.
`public`	Variable ist überall dort erreichbar, wo auch die Klasse erreichbar ist, zu der sie gehört.
`private`	Variable ist nur innerhalb der eigenen Klasse erreichbar.
`protected`	Variable ist nur innerhalb der eigenen Klasse und des Package erreichbar, zu dem sie gehört. Unterklassen können ebenfalls zugreifen.
`final`	Wert der Variable ist nicht veränderbar (Konstante).
`transient`	Inhalt der Variablen wird bei einer Serialisierung ignoriert.
`static`	Variable ist eine Klassenvariable, d.h. wird nicht instantiiert.

3.7.3 Modifikatoren für Methoden

Modifikator	Bedeutung
keiner	Methode ist innerhalb der eigenen Klasse und des Package erreichbar, zu dem sie gehört.
`abstract`	Methode besitzt zur Kopf und keinen Körper. Dieser muss von einer Unterklasse implementiert sein. Klasse muss ebenfalls als abstrakte Klasse definiert werden.
`public`	Methode ist von überall erreichbar.
`private`	Methode ist nur innerhalb der eigenen Klasse erreichbar
`protected`	Methode ist innerhalb der eigenen Klasse und des Package erreichbar, zu dem sie gehört. Unterklassen können ebenfalls zugreifen.
`final`	Methode kann von Unterklassen nicht überschrieben werden.
`static`	Methode ist eine Klassenmethode, d.h. wird nicht instantiiert.
`native`	Methode ist in einer anderen Programmiersprache realisiert. Es wird wie bei abstract nur der Methodenkopf definiert.