5 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:

5.1 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

5.2 Klassen und Objekte

Man unterscheidet Klassen und die eigentlichen Objekte (Instanzen).

5.2.1 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.

5.2.2 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.

5.2.3 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
    }
}
```

5.3 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);
    }
}
```

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);
    }
}
```

5.4 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:

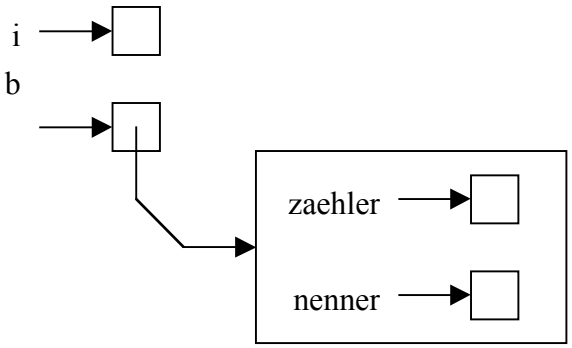
Deklaration:

```
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!

5.4.1 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
```

5.4.2 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.

5.5 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 5.2.1)

```
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.

5.5.1 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.

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).

5.5.2 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();
}
```

5.5.3 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 5.2.1)

```
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.

5.5.4 Mehrfachvererbung vs. Interfaces

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

Beispiel:

1. Die Thread-Anwendung aus Abschnitt 5.5 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.

5.5.4.1 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 5.5 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.

5.5.4.2 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:

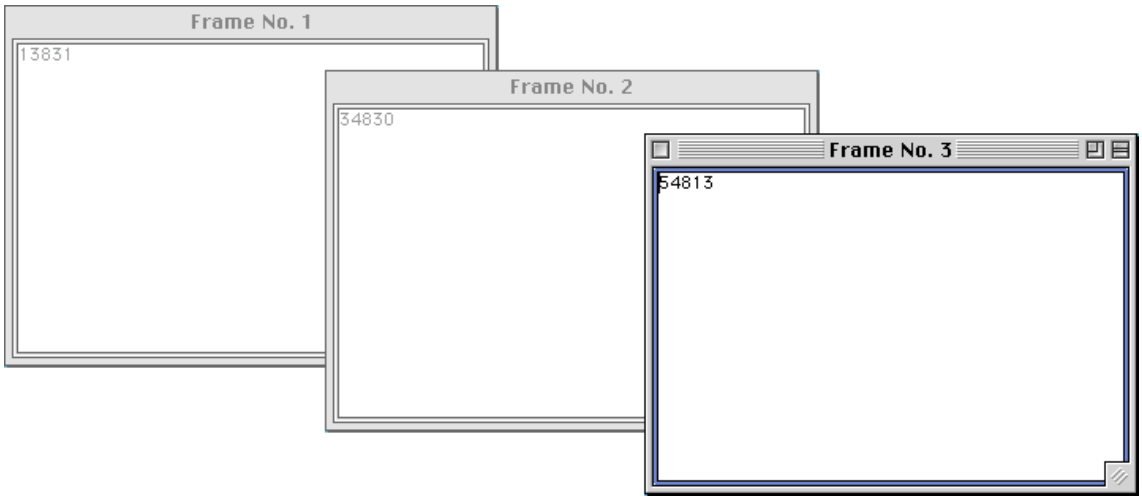
Es wird mit dem Schlüsselwort `implements` angezeigt, welche Interfaces zu implementieren sind: (siehe auch Syntax in Abschnitt 5.2.1)

```
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.



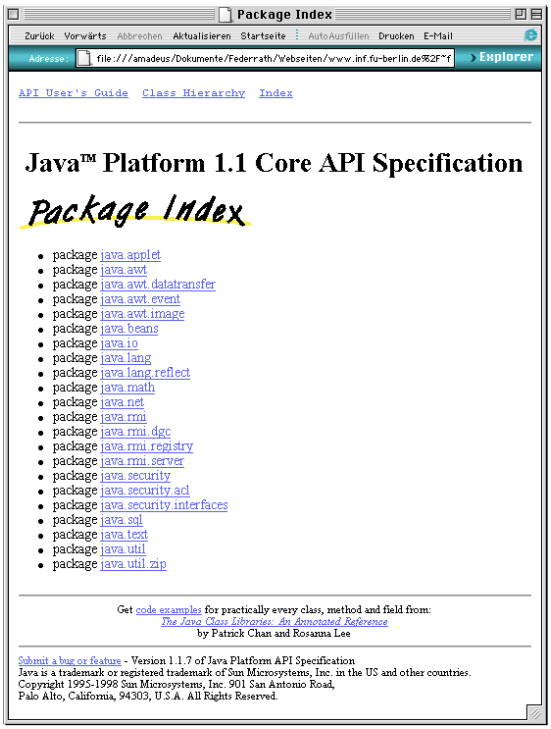
Sehr häufige Verwendung von Interfaces: Event-Handling bei graphischen Benutzeroberflächen.

5.6 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();
```

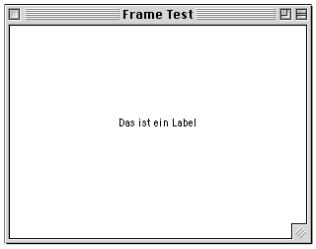
Importieren jeder einzelnen Klasse

```
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.

5.7 Modifikatoren in Java (Zusammenfassung)

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

5.7.1 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.

5.7.2 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.

5.7.3 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.

5.8 Oberflächenprogrammierung in Java – eine kleine Einführung

- Graphical User Interface (GUI)
- intensive Anwendung der objektorientierten Programmierung
- Nutzung der Java-Klassenbibliotheken

Es existieren momentan standardmäßig zwei verwendbare Klassenbibliotheken

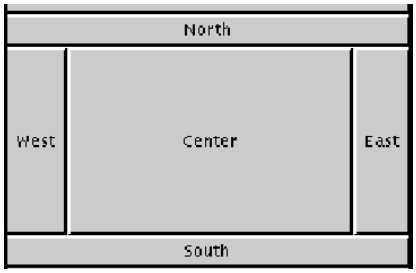
- Abstract Window Toolkit (AWT)
- Java Foundation Classes (JFC)
    - auch Swing-Bibliothek genannt
    - baut auf AWT auf
    - ab Java 1.2 Standard
    - nutzbar in Java 1.1 (muss separat heruntergeladen und installiert werden: http://www.javasoft.com/products/jfc

Im folgenden wird von JFC ausgegangen. Vorteile:

- deutlich vereinfachte Fensterprogrammierung gegenüber AWT
- plattformunabhängige Erscheinungsform (Look-and-Feel) von Java-Programmen
- bei Bedarf umschaltbar auf plattformabhängigen Look-and-Feel
- vorgefertige Klassen und Methoden für Dialoge
    - File öffnen, speichern: `JFileChooser()`
    - Warnungen und Fehlermeldungen: `JOptionPane.showMessageDialog()`

Die wichtigsten GUI-Elemente (Klassen)

- Normales Fenster: `JFrame`
- Menüleiste: `JMenuBar`
- Menüelement: `JMenuItem`
- Bedienelemente:
  - `JButton`: Knopf
  - `JCheckBox`: Auswahl☐
  - `JLabel`: nicht editierbares (einzeiliges) Textfeld
  - `JTextField`: editierbares (einzeiliges) Textfeld
  - `JTextArea`: editierbarer (mehrzeiliger) Textbereich
- Klassen zur Zusammenfassung von Bedienelementen
  - `JPanel`
- Anordnung der Bedienelemente in Layouts (vorgefertigte und ggf. eigene)
  - **BorderLayout:**



  - **FlowLayout** (GUI-Objekte werden nebeneinander angeordnet)

  - **GridLayout**

  - CardLayout
  - GridBagLayout (sehr universell, aber kompliziert zu programmieren)

Oberfläche mit Leben füllen:

- Auswertung von Events mittels Listener
  - `java.awt.event`
- Anwendung des Interface-Konzepts:
  - Z.B. `java.awt.event.ActionListener` stellt Schnittstelle zur Auswertung einer Action (z.B. Mausklick auf bestimmtes Bedienelement bereit:

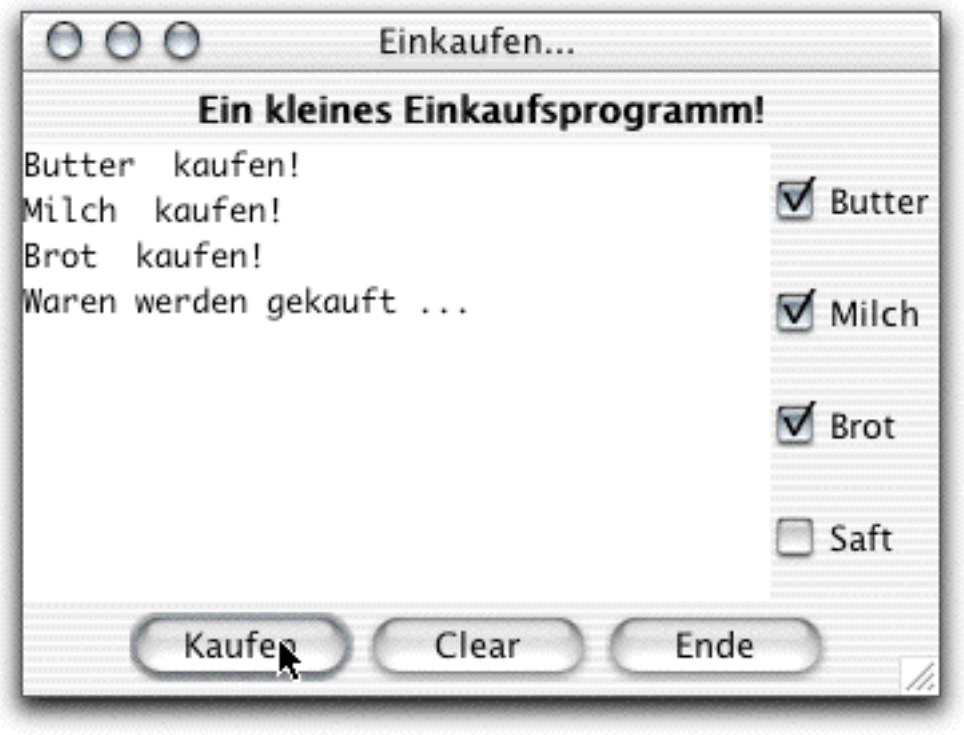
```
public abstract void actionPerformed(ActionEvent e)
```

Beispiel:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.UIManager;

public class Einkaufen extends JFrame implements ActionListener {
	public static final boolean DEBUG = true;

	public static final Font bigFont = new Font("Sans", Font.BOLD, 14);

	// Definition der ansprechbaren GUI-Elemente
	JTextArea ta = new JTextArea(10,40);
	JButton b1 = new JButton("Kaufen");
	JButton b2 = new JButton("Clear");
	JButton b3 = new JButton("Ende");
	JCheckBox c1 = new JCheckBox("Butter ");
	JCheckBox c2 = new JCheckBox("Milch");
	JCheckBox c3 = new JCheckBox("Brot");
	JCheckBox c4 = new JCheckBox("Saft");

	// Konstruktor
	public Einkaufen() {
		super("Einkaufen...");

		// Eventhandler für Fenster registrieren
		// (hier als "inner class" implementiert)
		addWindowListener( new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				beendeProgramm();
			}
		});

		// Panel-Objekt zur Anordnung eines Textlabels erzeugen
		JPanel p1 = new JPanel();
		JLabel l1 = new JLabel("Ein kleines Einkaufsprogramm!");
		l1.setFont(bigFont);
		// Panel besitzt implizit das FlowLayout
		p1.add(l1);

		// Panel-Objekt zur Anordnung der Buttons erzeugen
		JPanel p2 = new JPanel();
		// Panel besitzt implizit das FlowLayout
		p2.add(b1);
		p2.add(b2);
		p2.add(b3);
		// Eventhandler registrieren (=lokale Klasse)
		b1.addActionListener(this);
		b2.addActionListener(this);
		b3.addActionListener(this);

		// Panel-Objekt zur Anordnung der Checkboxen erzeugen
		JPanel p3 = new JPanel();
		// Layout auf GridLayout mit 4 Zeilen und 1 Spalte setzen
		p3.setLayout( new GridLayout(4,1) );
		p3.add(c1);
		p3.add(c2);
		p3.add(c3);
		p3.add(c4);
		// Eventhandler registrieren (=lokale Klasse)
		c1.addActionListener(this);
		c2.addActionListener(this);
		c3.addActionListener(this);
		c4.addActionListener(this);

		// Elemente zum Frame hinzufügen
		// Frame besitzt implizit das BorderLayout
		getContentPane().add(p1, BorderLayout.NORTH);
		getContentPane().add(p2, BorderLayout.SOUTH);
		getContentPane().add(p3, BorderLayout.EAST);
		getContentPane().add(ta, BorderLayout.CENTER);

		// Optimale Größe des Fensters setzen
		pack();
		// Fenster sichtbar machen
```

```
                setVisible(true);
        }

        // zu implementierende Methode der Schnittstelle ActionListener
        public void actionPerformed(ActionEvent event) {
                if(DEBUG)System.out.println("Event: " + event.getSource());
                // Auswerten der Events
                if (event.getSource() == c1)
                        ta.append("Butter " +
                                (c1.isSelected()?"":"NICHT") + " kaufen! \n" );
                else if (event.getSource() == c2)
                        ta.append("Milch " +
                                (c2.isSelected()?"":"NICHT") + " kaufen! \n" );
                else if (event.getSource() == c3)
                        ta.append("Brot " +
                                (c3.isSelected()?"":"NICHT") + " kaufen! \n" );
                else if (event.getSource() == c4)
                        ta.append("Saft " +
                                (c4.isSelected()?"":"NICHT") + " kaufen! \n" );
                else if (event.getSource() == b1) {
                        if (c1.isSelected() || c2.isSelected() ||
                            c3.isSelected() || c4.isSelected())
                                ta.append("Waren werden gekauft ... \n");
                        else
                                ta.append("Nichts auf der Einkaufsliste! \n");
                }
                else if (event.getSource() == b2) {
                        c1.setSelected(false);
                        c2.setSelected(false);
                        c3.setSelected(false);
                        c4.setSelected(false);
                        ta.append("Einkaufsliste wird geleert \n");
                }
                else if (event.getSource() == b3)
                        beendeProgramm();
        }

        public void beendeProgramm() {
                System.exit(0);
        }

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

Ausgabe:



5.9 Netzwerkprogrammierung

- Die wichtigsten Packages:
  - java.net für Sockets, URL-Handling
  - java.io für Ein-Ausgabe
- Im folgenden wird nur Behandlung von TCP vorgestellt.

5.9.1 Programmierung eines Clients

Beispiel: Abruf einer URL

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;

class GET {

	static void main(String[] argv) {
		// check for command line
		if (argv == null || argv.length != 1) {
			System.err.println("Usage: GET <url>");
			System.exit(1);
		}
		String url_string = argv[0];
		String line_read;
		try {
			// open the reader
			BufferedReader in = new BufferedReader (
				new InputStreamReader(
					(new URL(url_string)).openStream()
				)
			);
			// read input line by line
			while( (line_read = in.readLine()) != null ) {
				System.out.println(line_read);
			}
			in.close();
		} catch (Exception e) {
			System.err.println("Error " + e);
		}
	}

}
```

Ausgabe: (unter Verwendung des unten erläuterten Webservers)

```
> java GET http://localhost:8000/link.html
<html>
<title>It works</title>
<p>It works!</p>
<hr>
<address>ALP2 is fun!</address>
</html>
>
```

5.9.2 Programmierung eines Servers

- Klasse `ServerSocket` wartet auf eingehende Requests
- Jeder Request wird durch eine Instanz eines ConnectionHandlers bedient
- Quasi-Parallele Abarbeitung der ConnectionHandler durch Threads

Beispiel: Implementierung eines sehr kleinen Ausschnitts des Internet Standards RFC 2068 Hypertext Transfer Protocol -- HTTP/1.1

Zu implementierendes Protokoll: (□ steht für CRLF `"\r\n"`)

- Client □ Server:

```
GET http://www.domain.com/path/file.html HTTP/1.1 □
□
```

- Server □ Client:

```
HTTP/1.0 200 OK □
Content-type: text/html □
Content-length: <nr_of_bytes> □
□
<content>
```

Alternative Möglichkeiten f. Response:

```
HTTP/1.0 404 Not found
```

```
HTTP/1.0 400 Bad request
```

Achtung! Das ist nur ein sehr kleiner Ausschnitt von des HTTP!

Weitere Anforderungen:

- Die Konfiguration des Servers soll aus einer .properties-Datei geladen werden.
- Der Server soll Teil des Package `alp2` sein.

Es werden 2 Klassen implementiert:

- Klasse `Webserver` enthält `main`-Methode
- Klasse `ConnectionHandler`

Aufruf: `java alp2.Webserver <properties-File>`

Properties-File: (test.properties)

```
############################################################
# ALP2Webserver properties file
############################################################
# Options to define by the user:
#  localPort : local port where server listens
#  htdoc     : directory for file to be served
#  defaultfn : default filename, if a directory
#              ".../" is given only
#  debug     : enabled | disabled
############################################################
localPort  = 8000
htdoc      = alp2/htdoc
defaultfn  = index.html
debug      = enabled

############################################################
# Mapping table for file extensions to MIME types
############################################################
.txt   = text/plain
.html  = text/html
.htm   = text/html
.gif   = image/gif
.shtml = text/html
.jpg   = image/jpeg
.xml   = text/xml

############################################################
# HTML messages for error handling
############################################################
```

```
invalidRequest = <HTML><TITLE>400 Bad Request</TITLE><H1>400 Bad Request</H1>
                 <P>Your request has been rejected by the server.</HTML>
notFound       = <HTML><TITLE>404 File Not Found</TITLE><H1>404 File Not
                 Found</H1><P>File not found on this server.</HTML>

############################################################
# Header messages for error handling
############################################################
httpOK       = HTTP/1.0 200 OK
httpNOTFOUND = HTTP/1.0 404 Not found
httpERROR    = HTTP/1.0 400 Bad request
```

Quellcodes:

1. File: Webserver.java

```
package alp2;

import java.net.ServerSocket;
import java.net.Socket;
import java.util.ResourceBundle;

public class Webserver implements Runnable  {
	private boolean debug = false;
	private String rscFileName;
	private ResourceBundle msg;
	private int localPort = -1;
	private boolean runFlag;
	private Socket socket;
	private ServerSocket server;

	public Webserver(String ressourceFilename) {
		this.rscFileName = ressourceFilename;
		Thread t = new Thread(this);
		t.start();
	}

	public void run() {
		runFlag = true;

		// open resource (config) file
		try {
			msg = ResourceBundle.getBundle(rscFileName);
		} catch (Exception e) {
			System.err.println("Error: Cannot find resource file "
				+rscFileName+".properties");
			runFlag = false;
		}

		// load settings from ressource file
		if (runFlag) {
			debug = (this.getString("debug").equals("enabled")?true:false);
			localPort = Integer.parseInt(this.getString("localPort"));
		}

		// run server while runFlag is true
		System.out.println("Server on port "+localPort+" starting...");
		while (runFlag) {
			server = null;
			try {
				server = new ServerSocket (localPort);
				while(runFlag) {
					socket = server.accept();
					ConnectionHandler doIt =
						new ConnectionHandler (socket, this);
					Thread thread = new Thread (doIt);
					thread.start();
				}
			} catch (Exception e) {
				try {
					socket.close();
					server.close();
				} catch (Exception e2) {
					;
				}
				if(debug) System.err.println("Server Exception: "+e);
			}
```

```
                }
                System.out.println("Server on port "+localPort+" stopped!");
        }

        public void stopServer() {
                runFlag = false;
                try {
                        socket.close();
                        server.close();
                }
                catch (Exception e) {
                        ;
                }
        }

        public String getString(String s) {
                String ret;
                try {
                        ret = msg.getString(s);
                }
                catch (Exception e) {
                        System.out.println("Warning: Ressource \""+s+"\" not found!");
                        ret = "";
                }
                return ret;
        }

        public static void main(String[] argv) {
                // check for command line
                if (argv == null || argv.length != 1) {
                        System.err.println("Usage: alp2.Webserver <configfile>");
                        System.exit(1);
                }
                new Webserver(argv[0]);
        }
}
```

2. File: ConnectionHandler.java

```
package alp2;

import java.net.Socket;
import java.io.File;
import java.io.FileInputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.OutputStreamWriter;
import java.io.FileReader;
import java.util.ResourceBundle;

class ConnectionHandler extends Thread {
        static final String   CRLF="\r\n";
        private static int    threadCount;
        private int           threadNumber;
        private boolean       debug;
        private Socket        s;
        private Webserver     rsc;

        public ConnectionHandler(Socket s, Webserver rsc) {
                this.s = s;
                this.rsc = rsc;
                this.debug = (rsc.getString("debug").equals("enabled")?true:false);
        }

        public void run() {
                threadNumber = getThreadNumber();
                if(debug) System.out.println("Connection ("+threadNumber+") - New connection.");
                try {
                        // open connection streams
                        DataInputStream fromClient = new DataInputStream(s.getInputStream());
                        BufferedWriter  toClient   = new BufferedWriter(
                                new OutputStreamWriter(s.getOutputStream()));

                        // read and analyze first header line
                    String requestLine = this.readLine(fromClient) + "  ";
```

```
if(debug) System.out.println("Connection ("+threadNumber+
                ") - Header: " + requestLine + "");
int firstSpacePosition = requestLine.indexOf(' ');
String method = requestLine.substring(0,firstSpacePosition);
String url = requestLine.substring(firstSpacePosition + 1);
// cut the protocol String from the file if existing
firstSpacePosition = url.indexOf(' ');
url = url.substring(0,firstSpacePosition);

if (method.equalsIgnoreCase("GET")) {
        // request a file from the server
        String fn = url.substring(1);
        // change the http file/dir separator '/' to the system specific one
        fn.replace('/',File.separatorChar);
        if (url.endsWith("/"))
                fn += rsc.getString("defaultfn");
        String path = rsc.getString("htdoc");
        // create file object
        File file = new File(path,fn);
        if (file.exists() && file.isFile() && file.canRead()) {
                // process request
                toClient.write(rsc.getString("httpOK")+CRLF);

                // find and send Content-type
                String ext = null;
                String typ = null;
                int idx = fn.lastIndexOf(".");
                if (idx != -1 ) {
                        ext = fn.substring(idx).toLowerCase();
                        typ = rsc.getString(ext);
                }
                toClient.write("Content-type: "+typ+CRLF);

                // get file length for header Content-length
                long len = file.length();
                if (len != 0L)
                        toClient.write("Content-length: "+len+CRLF);

                // send final CRLF
                toClient.write(CRLF);

                // send data
                int b;
                try {
                        BufferedInputStream fromFile =
                                new BufferedInputStream(
                                new FileInputStream(file));
                        // flush buffer before opening a new stream
                        // to the same output
                        toClient.flush();
                        // open data output stream (needed to transfer
                        // data without changes)
                        BufferedOutputStream toClientX =
                                new BufferedOutputStream(
                                new DataOutputStream(s.getOutputStream()));
                        while ( (b = fromFile.read()) != -1 ) {
                                toClientX.write(b);
                        }
                        fromFile.close();
                        toClientX.flush();
                        toClientX.close();
                }
                catch (Exception e) {
                        System.out.println("Connection ("+threadNumber+
                                ") - File error: "+e);
                }
        } else {
                // send "not found"
                toClient.write(rsc.getString("httpNOTFOUND")+CRLF);
                toClient.write("Content-type: "+rsc.getString(".html")+CRLF);
                toClient.write(CRLF);
                toClient.write(rsc.getString("notFound"));
        }
} else {
// output error message if request method does not match
        toClient.write(rsc.getString("httpERROR")+CRLF);
        toClient.write("Content-type: "+rsc.getString(".html")+CRLF);
        toClient.write(CRLF);
        toClient.write(rsc.getString("invalidRequest"));
```

```
                }
                toClient.flush();
                toClient.close();
                fromClient.close();
                s.close();
        }
        catch (Exception e) {
                if(debug) System.out.println("Connection ("+threadNumber+
                                ") - Exception: "+e);
        }
        if(debug) System.out.println("Connection ("+threadNumber+
                        ") - Connection closed.");
    }

    private String readLine(DataInputStream inputStream) throws Exception {
        String returnString = "";
        try{
                int b = inputStream.read();
                while (b != 10 && b != -1) {
                        if (b != 13)
                                returnString += (char)b;
                        b = inputStream.read();
                }
        } catch (Exception e) {
                throw e;
        }
        return returnString;
    }

    private synchronized int getThreadNumber() {
        return threadCount++;
    }

}
```