3 Das Projekt Bankkonto

Nun wirst du dich etwas gründlicher mit dem Quelltext einer Klasse beschäftigen. Du lernst, wie zwei Objekte eine gemeinsame Aufgabe erledigen.

Bemerkung:
Das Projekt *Bankkonto* stellt Herr **Prof. Dr. H. Peter Gumm** von der Universität Marburg auf seinen Webseiten den Studenten zur Verfügung. Ich habe diese Idee übernommen und sie an meinen Unterricht angepasst.

Das neue Projekt ist eine naive Implementierung eines Bankkontos. Du erinnerst dich noch daran, dass in den Programmen nur ein Modell eines Ausschnitts der realen Welt erzeugt wird. Deswegen wirst du schnell merken, dass die Implementierung in diesem Kapitel noch viele Schwächen hat.

3.1 Erstellung eines Modells

Bevor du ein Konto in ein Programm umsetzen kannst, musst du dir darüber Gedanken machen, welche Eigenschaften das Konto haben soll und was man mit diesem Konto alles machen kann. Du entwirfst also ein Modell eines Kontos. Dazu helfen dir die Antworten auf folgende Fragen:

Was ist ein Konto? (Eigenschaften)

a) Ein Konto hat eine Nummer.
b) Ein Konto speichert den Namen des Inhabers.
c) Ein Konto speichert einen aktuellen Kontostand.

Was kann man mit einem Konto machen? (Fähigkeiten)

a) Von einem Konto kann man den aktuellen Kontostand abfragen.
b) Auf ein Konto kann man einen Geldbetrag einzahlen.
c) Von einem Konto kann man einen Geldbetrag abheben.
d) Auf ein anderes Konto kann man einen Geldbetrag überweisen.

Das Modell des Kontos in Abbildung 3.1 zeigt also, dass die Klasse *Konto* die drei Datenfelder *nummer*, *inhaber* und *kontostand* besitzt. Zusätzlich verfügt sie über die Methoden *gibKontostand()*, *einzahlen()*, *abheben()* und *ueberweisen()*.

Abbildung 3.1: Modell eines Kontos

3.2 Implementieren der Klasse Konto

Dieses Modell wird nun in einen Quelltext übersetzt. Diesen Vorgang bezeichnet man als *Implementieren*. Du implementierst nun eine Klasse *Konto*, die als Fabrik bzw. als Vorlage für beliebig viele Konten dient.

3.2.1 Datenfelder beschreiben die Eigenschaften

Abbildung 3.2 zeigt drei Objekte *felixKonto*, *veraKonto* und *ninaKonto* der Klasse *Konto*. Alle diese Objekte besitzen die gleichen Datenfelder *nummer*, *inhaber* und *kontostand*, unterscheiden sich aber in den Werten, die in diesen Datenfeldern stehen.

Abbildung 3.2: Objekte der Klasse *Konto* mit ihren Datenfeldern

Abbildung 3.3 zeigt den zugehörigen Quelltext.

```
/**
 * Die Klasse Konto modelliert ein einfaches Bankkonto.
 *
```

```
 * @author Ralph Henne
 * @version 27.9.03
 */
class Konto
{
    int nummer;
    String inhaber;
    int kontostand;
}
```

Abbildung 3.3: Quelltext der Klasse *Konto* enthält die Datenfelder

Übung 3.1:

a) Erstelle in **BlueJ** ein neues Projekt *Bankkonto*.

b) Mit *New Class* erstelle nun innerhalb dieses Projekts eine neue Klasse *Konto*.

c) Ersetze im Editorfenster den gesamten vorgeschlagenen Quelltext durch den in Abbildung 3.3 dargestellten.

d) Kompiliere die Klasse *Konto*.

e) Erzeuge mit *new Konto()* die drei Konten *felixKonto*, *veraKonto* und *ninaKonto*.

f) Untersuche mit Hilfe des Inspektors das *felixKonto*.

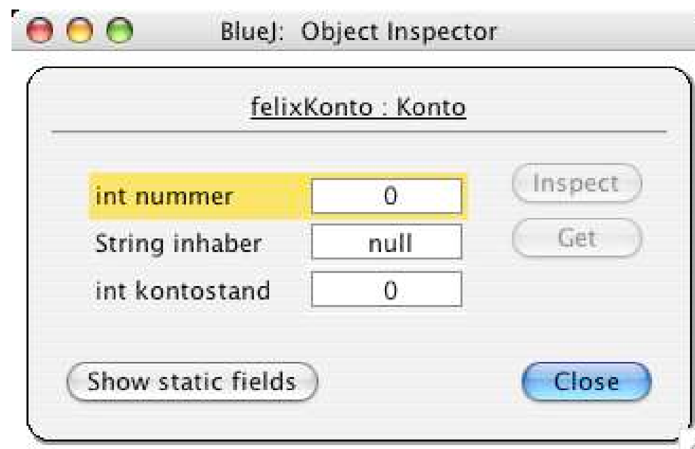


Abbildung 3.4: Inspektor des Objekts *felixKonto*

Der Inspektor aller Konten zeigt, dass die Datenfelder *nummer* und *kontostand* mit *0* und *inhaber* mit *null* belegt sind.

3.2.2 Konstruktoren initialisieren Objekte

Abbildung 3.4 zeigt, dass die Datenfelder *nummer*, *inhaber* und *kontostand* des Objekts *felixKonto* mit den sogenannten Standardwerten belegt sind. Allerdings

sollten diese Datenfelder jedoch bereits bei der Erzeugung des Objekts mit Werten belegt. Um Objekte mit den richtigen Initialisierungen zu erzeugen, werden *Konstruktoren* benötigt.

Ein Konstruktor ist eine besondere Methode, die den gleichen Namen wie die Klasse besitzt. Sie wird automatisch und einmalig bei der Erzeugung des Objekts aufgerufen und setzt dieses in einen sinnvollen Anfangszustand.

```
/** Konstruktoren */
public Konto()
{
    nummer = 1234;
    inhaber = "felix";
    kontostand = 100;
}
```

Abbildung 3.5: Möglicher Konstruktor der Klasse *Konto*

Übung 3.2:

Implementiere den Konstruktor in der Klasse Konto.

a) Erzeuge ein Objekt *felixKonto* und überprüfe die Attributwerte mit Hilfe des Inspektors.

b) Erzeuge ein weiteres Objekt *ninaKonto* und überprüfe auch dieses mit Hilfe des Inspektors.

Du kannst nun in die Datenfelder Werte hineinschreiben. Unsinnigerweise besitzen jedoch alle Objekte der Klasse *Konto* dieselben Werte. Du solltest also einen Konstruktor implementieren, der dich nach der Nummer und dem Namen des Inhabers fragt.

```
public Konto(int nummerH, String inhaberH)
{
    nummer = nummerH;
    inhaber = inhaberH;
    kontostand = 0;
}
```

Abbildung 3.6: Besserer Konstruktor der Klasse *Konto*

Übung 3.3:

Implementiere den in Abbildung 3.6 dargestellten Konstruktor.

a) Erzeuge nun die Objekte *felixKonto* und *ninaKonto* und überprüfe diese mit Hilfe des Konstruktors. Vergleiche hierzu Abbildung 3.7.

b) Du musst die Funktionsweise diese Konstruktors verstanden haben.

Formuliere diese schriftlich!

Bemerkung:
Ich bezeichne im Konstruktor die formalen Parameter mit dem Buchstaben „**H**“ für **H**ilfsparameter. Somit entfällt die Suche nach einem weiteren geeigneten Parameternamen. In der Literatur werden häufig Bezeichnungen wie *neuNummer* verwendet.

Eine weitere Möglichkeit ist die Verwendung des Schlüsselworts **this**. Jedoch hat die Einführung dieses Schlüsselworts zu diesem Zeitpunkt die Schüler nur verwirrt.

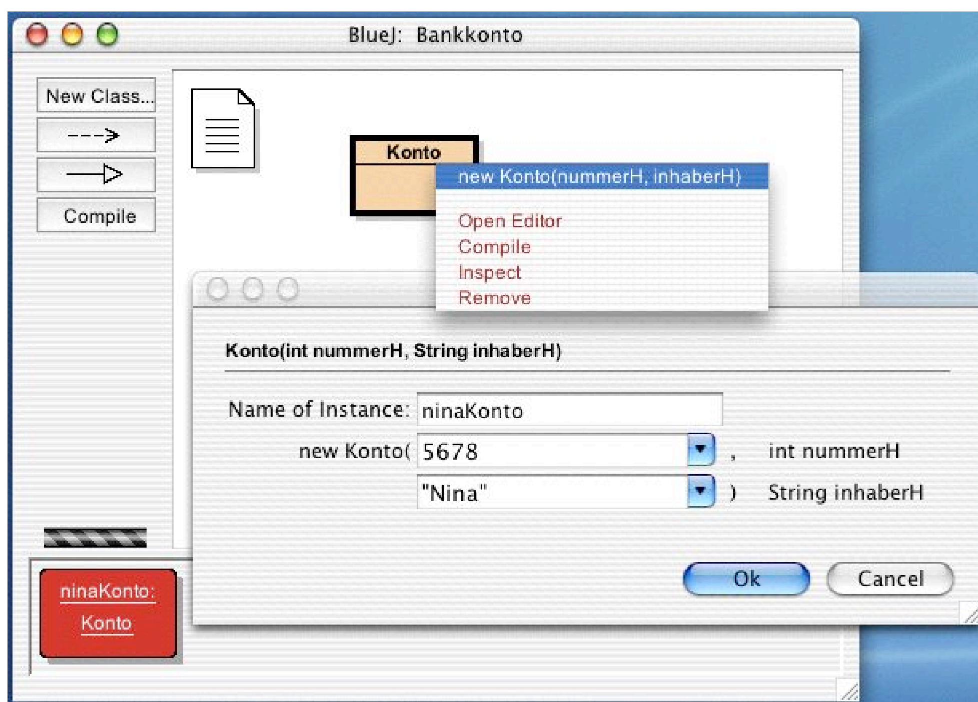


Abbildung 3.7: Erzeugung des Objekts *ninaKonto* mit Hilfe eines Konstruktors

Die Klasse *Konto* hat nun einen Konstruktor mit zwei Parametern. Bei der Erzeugung des Objekts werden die gewünschten Werte der Parameter abgefragt. Der Aufruf *new Konto (5678, „Nina“)* erzeugt also das Objekt *ninaKonto*.

Abbildung 3.8: Erzeugung eines Objekts *ninaKonto* mit Hilfe des besseren Konstruktors

Wie du bereits festgestellt hast, konntest du bereits ohne Implementierung eines Konstruktors Objekte erzeugen. Hier wird automatisch ein Standardkonstruktor aufgerufen, der im Prinzip folgendes Aussehen hat:

```
public Konto()
{ }
```

Abbildung 3.9: Standardkonstruktor

Da die Konstruktormethode leer ist, erhalten alle Datenfelder ihre Standardwerte. Die erzeugten Objekte hast du bereits in Übung 3.1 untersucht.

Eine Klasse kann mehrere Konstruktoren besitzen. So kannst du mit verschiedenen Konstruktoren folgende Situationen simulieren:

a) Eine Bank will einem Neukunden gleich ein Begrüßungsgeld überweisen.

b) Wenn ein Neukunde von einem anderen Kunden geworben wurde, soll der Werber eine Prämie erhalten.

Jedoch müssen alle verwendeten Konstruktoren sich in der Parameterliste unterscheiden. Anhand der Parameter wird der richtige Konstruktor bei der Ausführung der Methode *new()* aufgerufen.

In der Klasse *Konto* sind **gleichzeitig erlaubt**:

Konto()
Konto(int nummer, String name)
Konto(String name, int nummer)
Konto(int nummer, String name, int begruessungsGeld)
Konto(int nummer, String name, Konto werber)

In der Klasse *Konto* sind Parameter mit gleichen Typen **gleichzeitig verboten**:

Konto(int nummer, String name)
Konto(int betrag, String empfänger)

Übung 3.4:
Eine Bank will einem Neukunden gleich ein Begrüßungsgeld gutschreiben.
Implementiere einen geigneten Konstruktor.

Übung 3.5:
Implementiere die gleichzeitig erlaubten Konstruktoren und teste diese.
Implementiere auch die gleichzeitig verbotenen Konstruktoren und beachte die Fehlermeldung.

3.2.3 Methoden beschreiben die Fähigkeiten

Bis jetzt kannst du mit den Konten nicht viel anfangen. Konten werden verwendet, um Geld einzuzahlen bzw. abzuheben. Ab und zu willst du auch wissen, wie viel Geld du auf dem Konto liegen hast. Und zum bequemen Bezahlen von Rechnungen verwendest Du eine Überweisung.

Um die oben beschriebenen Situationen zu simulieren, musst du die Klasse *Konto* mit geeigneten Methoden ergänzen. Das Modell wird nun um die die Methoden *gibKontoStand()*, *einzahlen()*, *abheben()* und *ueberweisen()* erweitert.

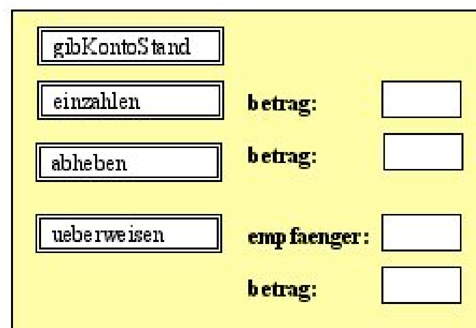


Abbildung 3.10: Methoden der Klasse *Konto*

Die Methoden lassen sich in zwei Arten klassifizieren:

Sondierende Methoden liefern Objekt-Informationen

> Sie liefern einen Wert zurück, z.B. eine Zahl, einen String, o. Ä.
> Sie sollen das Objekt **nicht** verändern.

Verändernde Methoden verändern das Objekt

> Sie verändern den Zustand eines Objekts.
> Sie werden i. A. nicht verwendet, um Informationen über das Objekt zu liefern.

Übung 3.6:

Abbildung 3.10 zeigt die Methoden des Modells *Konto*. Benenne die sondierenden und die verändernden Methoden.

Dieser Unterschied zeigt sich auch in der Implementierung der beiden Methoden:

```
/** Liefert den aktuellen Kontostand. */
public int gibKontostand()
{
    return kontostand;
}

/** Erhoeht den Kontostand um betrag. */
public void einzahlen(int betrag)
{
    kontostand += betrag;
}
```

Abbildung 3.11: Die Methoden *gibKontostand()* und *einzahlen()* in der Klasse *Konto*

Die Methode *gibKontostand()* zeigt den Wert des Datenfeldes *kontostand*. Sie liefert also einen Rückgabewert mit dem Datentyp *int*. Dieser Datentyp muss im Methodenkopf `public int gibKontostand()` beschrieben werden. Den Wert des Datenfeldes *kontostand* liefert die *return*-Anweisung.

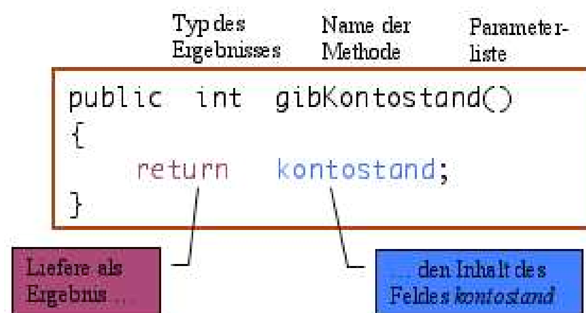


Abbildung 3.12: Sondierende Methode *gibKontostand()*

Die Methode *einzahlen()* besitzt keinen Rückgabewert. Auch dies wird im Methodenkopf `public void wechsleGang(int gang)` mit dem Schlüsselwort *void* (engl.: Leerstelle, Fehlstelle) dargestellt. Allerdings ändert diese Methode den Wert des Datenfeldes *kontostand*. Der neue Wert wird der Methode als Parameter `int betag` übergeben. Nun kann der Parameterwert in das Datenfeld *kontostand* eingelesen werden.

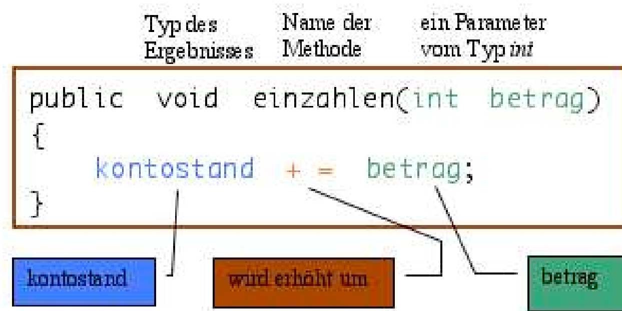


Abbildung 3.13: Verändernde Methode *einzahlen()*

Übung 3.7:

Implementiere in der Klasse *Konto* die beiden Methoden *gibKontostand()* und *einzahlen()* wie in Abbildung 3.11 dargestellt. Erzeuge anschließend ein Objekt *felixKonto*. Teste nun die beiden Methoden. Liefern sie die erwarteten Ergebnisse? Vergleiche dazu die Abbildung 3.14.

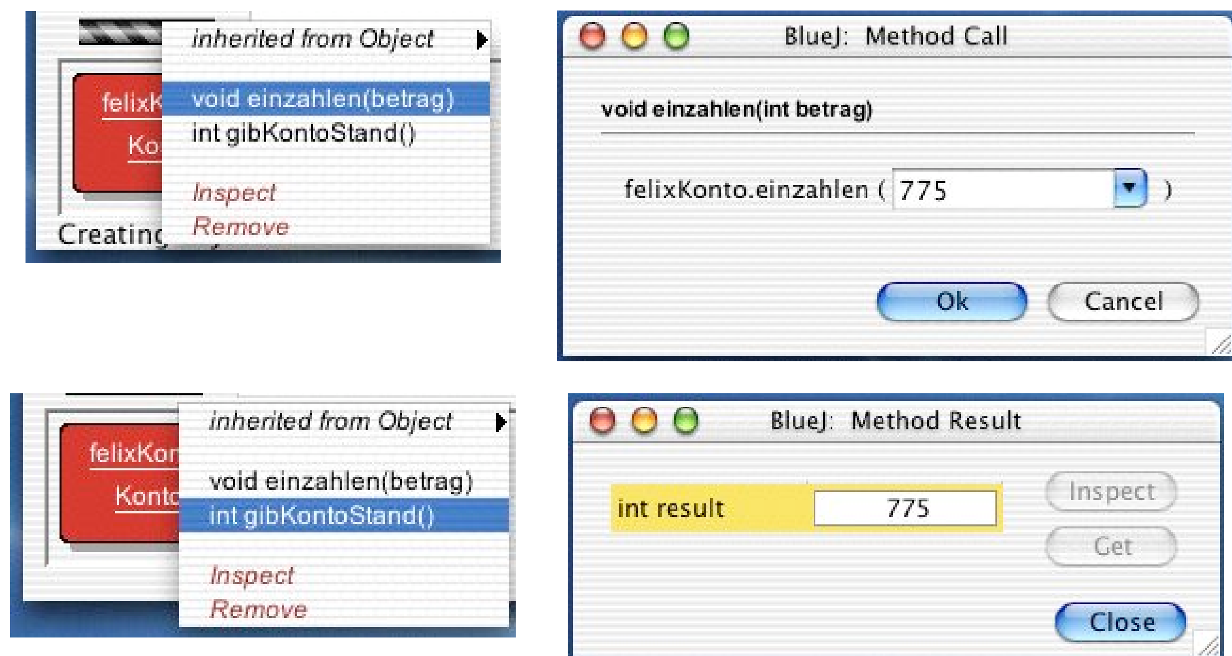


Abbildung 3.14: Die Methoden *einzahlen()* und *gibKontoStand()*

Übung 3.8:

Implementiere nun eine Methode *abheben()*, die einen bestimmten Betrag vom Konto abhebt. Teste deine neue Methode mit Hilfe des Inspektors!

Mit Hilfe des Inspektors lässt sich das Verhalten der Objekte sogar „live“ beobachten.

3.3 Die Methode „ueberweisen()“

Diese Methode ist etwas schwieriger. Hier kommunizieren nämlich zwei Objekte miteinander.

Was benötigt *ueberweisen()*?

> das Konto des Empfängers,
> den Betrag.

Was macht *ueberweisen()*?

> hebt den Betrag vom eigenen Konto ab,
> zahlt den Betrag auf das Empfänger-Konto ein.

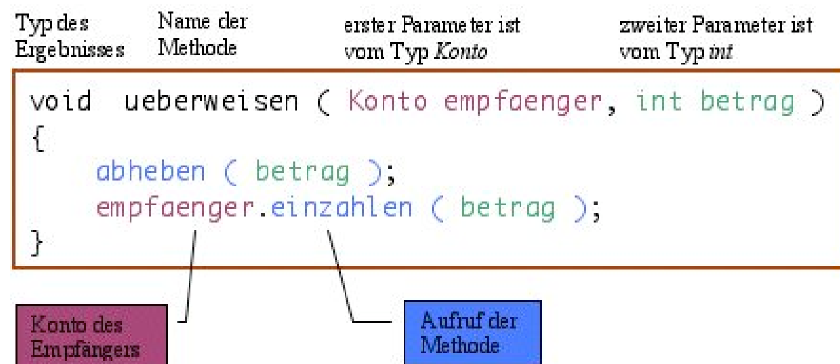


Abbildung 3.15: Die Methode ue*berweisen()*

Beim Aufruf der Methode *ueberweisen()* werden die **formalen Parameter** *empfaenger* bzw. *betrag* ersetzt durch die **konkreten Werte (aktuelle Parameter)** *veraKonto* bzw. *432*.

Der Aufruf

```
ueberweisen( veraKonto , 432 )
```

führt zu

```
abheben ( 432 );
veraKonto.einzahlen ( 432 ) ;
```

Abbildung 3.16: Wirkungen der Methode ue*berweisen()*

Hier liegt eine **Objekt-Kommunikation** vor:

Das Objekt, das die Methode *ueberweisen()* benutzt, in unserem Fall *felixKonto*, kommuniziert mit dem Objekt, das als Empfängerkonto benannt ist.

Zuerst ruft *felixKonto* seine eigene Methode *abheben()* auf. Dann erhält der Empfänger, hier *veraKonto*, die Weisung, dessen eigene Methode *einzahlen()* aufzurufen mit den Parameter *432*.

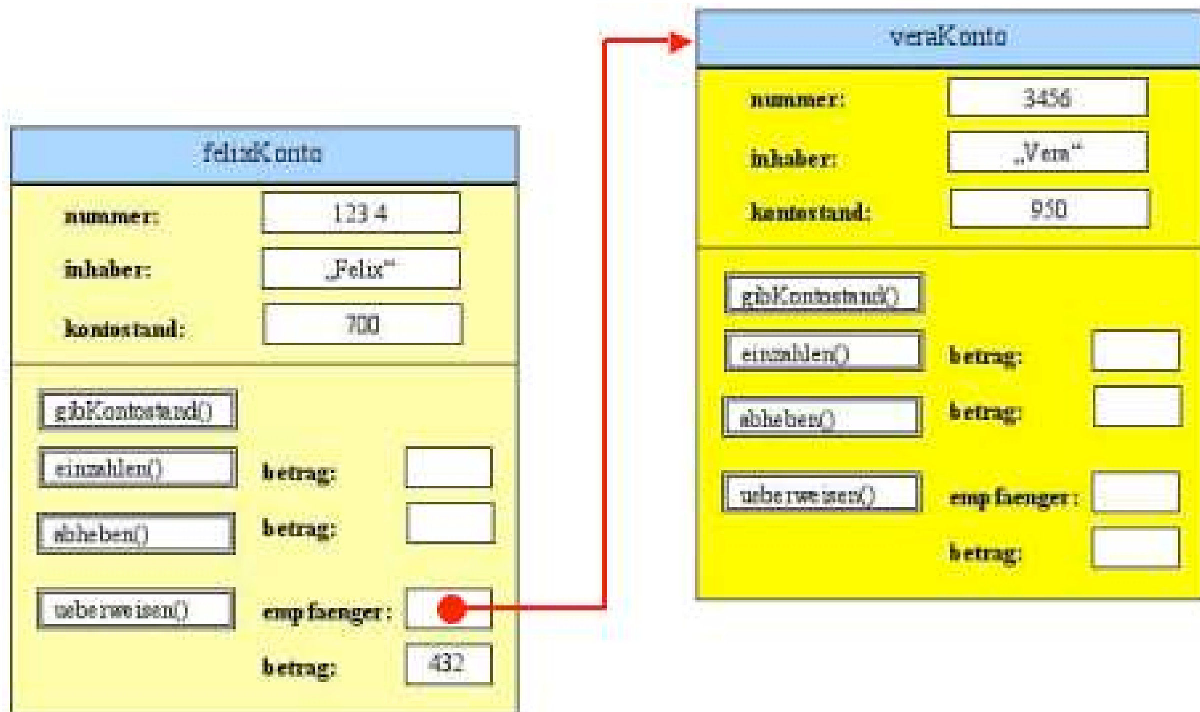


Abbildung 3.17: Grafische Darstellung der Methode *ueberweisen()*

Die Eingabe des Konto-Parameters in **BlueJ** erfolgt

durch Eingabe des Namens, hier *veraKonto*, oder
durch Klick auf das Objekt *veraKonto* in der Objektleiste

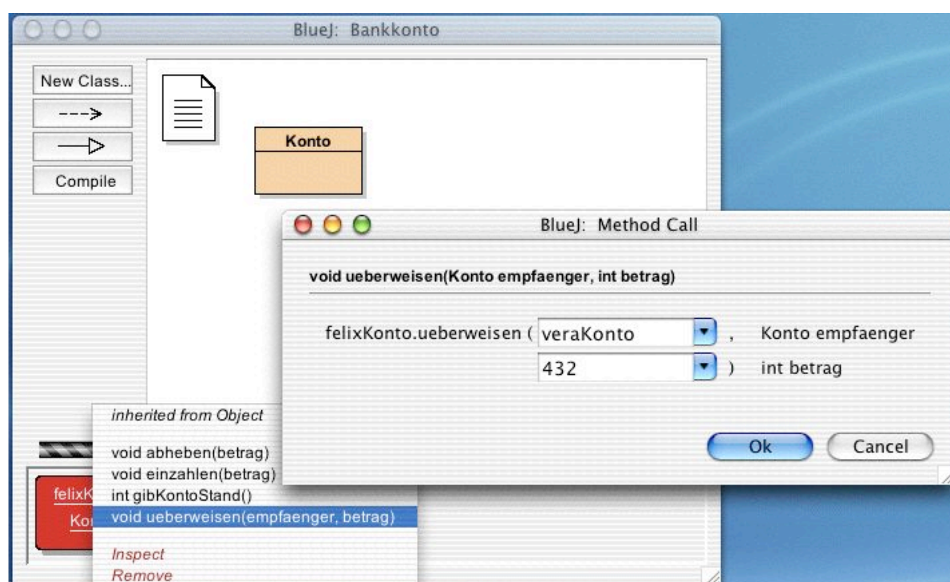


Abbildung 3.18: Darstellung der Methode ue*berweisen()* in **BlueJ**

Übung 3.9:

Implementiere nun die Methode *ueberweisen()*, die einen bestimmten Betrag von einem Konto auf ein anderes Konto überweist. Teste deine neue Methode mit Hilfe des Inspektors!

Übung 3.10: (für Experten)

Implementiere eine Methode *einziehen(),* die dem Bankeinzugs-Verfahren entspricht.

3.4 Dokumentation

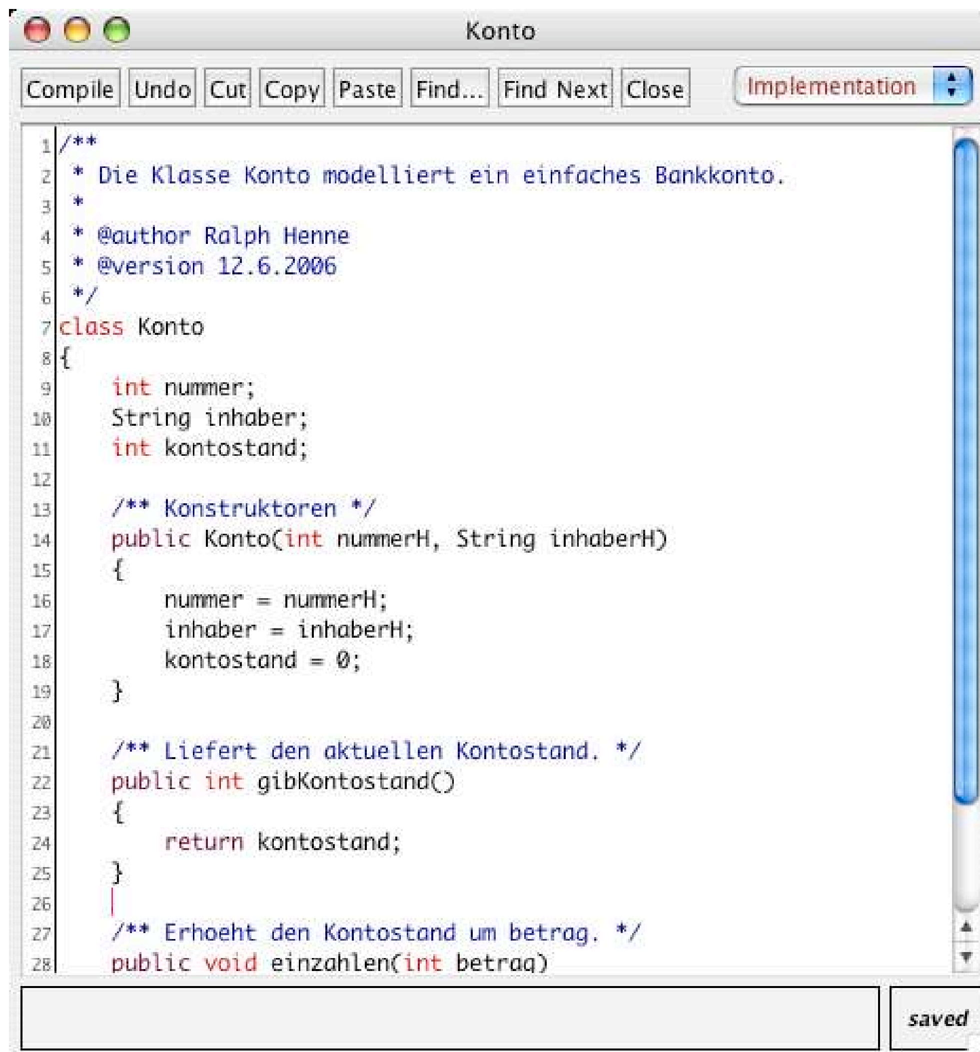


Abbildung 3.19: Klasse *Konto* mit Kommentaren im Editorfenster

Wichtiger Hinweis:
Vermeide sowohl im Quelltext als auch in der Dokumentation deutsche **Sonderzeichen**. Diese Sonderzeichen führen meistens zu Problemen bei der Ausführung der Programme auf anderen Plattformen.

Übung 3.11:

Dokumentiere den gesamten Quelltext der Klasse *Konto*.

1. Terminaufgabe:

In einer Weinkellerei werden in großen Fässern verschiedene Rotweinsorten gelagert. Diese sollen nun in weiteren Fässern gemischt werden.

Programmiere eine Klasse *Fass*. Überlege dazu

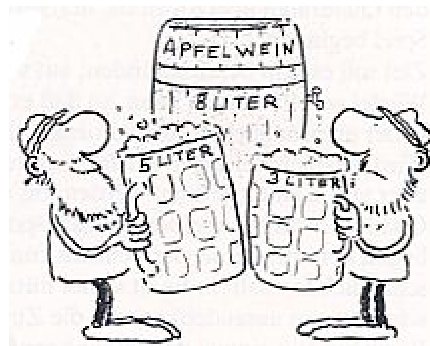
welche Eigenschaften ein Fass hat,
was man mit den Fässern machen kann.

Erstelle die entsprechenden Datenfelder und programmiere die benötigten Methoden.

Vergiss nicht, deinen Quelltext zu kommentieren.

2. Terminaufgabe:

Ein Bauer und sein Freund kauften ein 8-Liter-Fass, gefüllt mit besten Apfelwein. Sie wollten den Wein gerecht aufteilen, besaßen aber nur einen 5- und einen 3-Liter-Krug. Wie gelang es ihnen dennoch?



Spiele diese Aufgabe mit Deinem Programm durch!

Hinweis:
Du wirst schnell merken, dass deine Implementierung sehr viele Schwächen hat. Was passiert, wenn in den 3-Liter-Krug zum Beispiel 5 Liter Apfelwein gefüllt werden? Nenne weitere Schwächen! Hier fehlen uns noch einige Kontrollstrukturen. Wer bereits etwas Erfahrung in Programmiersprachen besitzt, kann versuchen, auch diese Mängel zu beheben.

Komprimiere den Ordner, in dem sich dein Projekt befindet und sende ihn an deinen Informatik-Lehrer unter folgender Adresse:

henne.info@gmx.de

Im Unterricht ergaben sich weitere Fragen zu diesem Kapitel. Diese werden besprochen im Zusatzkapitel „03zFass“