3 Objektorientierung – Grundbegriffe

Java 2

ISBN 3-8273-2028-3

Um Java programmieren zu können, ist es wichtig, einige objektorientierte Grundkenntnisse zu besitzen, denn die Sprache setzt voll auf dem OO-Paradigma auf.

3.1 Klassen und Objekte

Klassen

Klassen sind prinzipiell nichts weiter als benutzerdefinierte Datentypen, die ebenso verwendet werden wie die Basistypen *int, double* oder *char* bzw. Strukturen wie *struct* in C, *record* in Pascal oder *type* in Basic. Klassen sind Schablonen, also eine Art exakt definierter Baupläne, aus denen zur Laufzeit *Instanzen* erzeugt werden. Diese Instanzen bezeichnet man auch als *Objekte.* Eine Klassendefinition in Java beginnt mit dem Schlüsselwort *class* und dem Namen der Klasse. Im folgenden Block, symbolisiert durch geschweifte Klammern, werden dann die Bestandteile der Klasse definiert:

```
class Konto       // Klasse Konto
{
   int    nummer; // Variable
   double stand;  // Variable
}
```

Variablen

Den Inhalt jeder Klasse, also deren Eigenschaften (*Member*), bilden Daten und Funktionen. Die Daten jedes Objekts, also deren Attribute, sind in den *Variablen* oder *Feldern* der Klasse abgelegt. Da diese Variablen in jeder Instanz (= jedem Objekt) anders belegt sein können, nennt man sie meist *Instanzvariablen.*

Methoden

```
class Konto       // Klasse Konto
{
   int    nummer; // Variable
   double stand;  // Variable

   void einzahlen(double betrag)  // Methode
   {
      stand = stand + betrag;
   }
}
```

Das Verhalten der Klassen wird durch deren Funktionen beschrieben. In Java bezeichnet man die Funktionen einer Klasse als *Methoden*. Diese Methoden unterscheiden die Klassen von den Strukturen oder Records prozeduraler Sprachen wie C oder Pascal. In objektorientierten Sprachen kapselt ein Datentyp nicht nur den strukturierten Aufbau und den Zustand eines Objekts, sondern auch dessen Funktionalität. Die Methoden sind also nicht global einem bestimmten Programm zugeordnet, sondern gehören zu einer Klasse. Damit ist die Klasse auch nicht an eine bestimmte Anwendung gebunden, sondern steht allen Nutzern offen.

Meist gibt es noch eine besondere Art von Methoden, die so genannten *Konstruktoren*. Diese dienen dazu, neu erzeugte Objekte zu initialisieren.

```
class Konto
{
   int nummer;
   double stand;

   Konto(int neueNummer, double neuerStand) // Konstruktor
   {
      nummer = neueNummer;
      stand = neuerStand;
   }
}
```

Grafische Darstellung

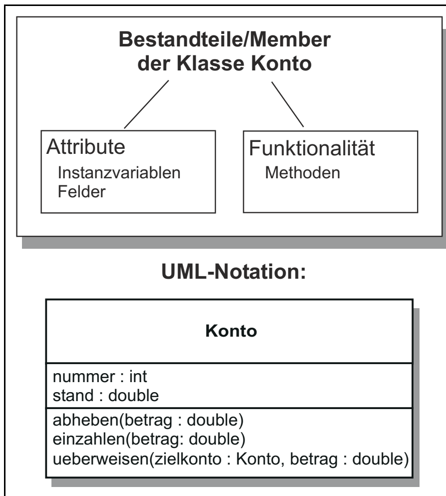

Bild 3.1: Bestandteile einer Klasse

In Abbildung 3.1 sehen Sie die Bestandteile einer Klasse grafisch dargestellt. Wir wollen im weiteren Verlauf des Buches – soweit möglich – die Notation der *Unified Modeling Language* (*UML*) benutzen, die sich für die Modellierung in objektorientierten Sprachen durchgesetzt hat. In der UML werden Klassen durch einen Kasten repräsentiert, oben steht der Klassenname, im mittleren Block folgen die Variablen und im unteren die Methoden.

Eine Übersicht über alle im Buch benutzten UML-Symbole finden Sie im Anhang A.

Objekte und Objektreferenzen

Aus dem Bauplan einer Klasse werden zur Laufzeit Objekte, also konkrete Ausprägungen, erzeugt. Dies macht man wie in C++ mit dem Schlüsselwort *new*. Ein Objekt bzw. eine Instanz gehört genau einer Klasse an und hat einen Zustand, nämlich die aktuelle Belegung der Felder. In der Regel gibt es von einer Klasse gleichzeitig mehrere Instanzen, deren Zustand voneinander unabhängig ist.

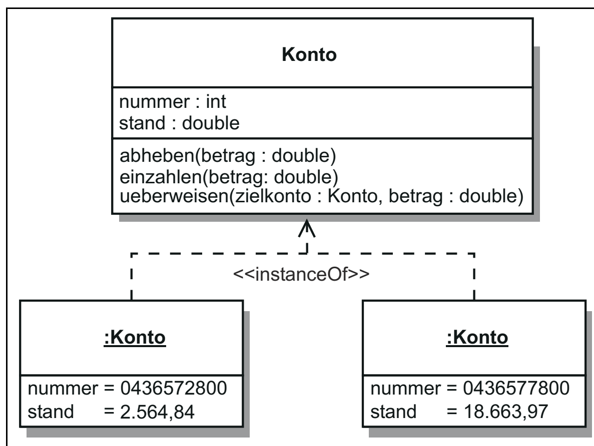

Bild 3.2: Zwei Instanzen der Klasse Konto

Abbildung 3.2 zeigt zwei verschiedene Konto-Objekte in der UML-Darstellung. Die gestrichelte Linie zeigt die Instanziierung an. Jede Instanz wird mit einem Doppelpunkt und durch Unterstreichung gekennzeichnet. Für die einzelnen Instanzvariablen wird der Wert angegeben.

Objekte werden im Speicher abgelegt, die automatische Speicherverwaltung von Java kümmert sich darum, dass genügend Speicherplatz angefordert wird und dieser bei Löschung des Objektes wieder freigegeben wird. Es nützt aber nichts, dass das Objekt irgendwo im Speicher existiert, man muss darauf auch wieder zugreifen können. Dazu verwendet man Referenzen, die auf ein bestimmtes Objekt verweisen. Man kann sich Objektreferenzen quasi als Namen des Objekts vorstellen (solange man dieser Referenz kein anderes Objekt zuweist). Referenzen sind also eine Art intelligente Zeiger, die automatisch immer auf das Objekt verweisen, auch wenn es sich im Speicher verschiebt.

Es handelt sich jedoch nicht um wirkliche Zeiger wie in C. Denn hinter den Referenzen stecken keine Speicheradressen und man kann auch keine Zeigerarithmetik betreiben!

Und so erzeugt man in Java ein Konto-Objekt und weist es einer Referenz zu:

```
Konto kundenkonto = new Konto();
```

Das erste *Konto* ist der Typ der Referenz mit dem Namen `kundenkonto`, diese Referenz kann also nur auf Instanzen der Klasse *Konto* verweisen. Diesem wird ein neues Objekt der Klasse *Konto* zugewiesen (siehe Abbildung 3.3).

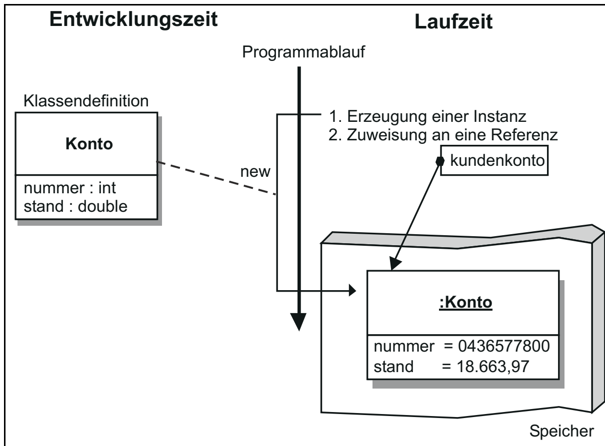

Bild 3.3: Erzeugen einer Instanz der Klasse Konto

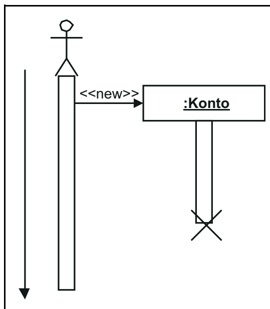

Bild 3.4: Instanziierung in UML

3

Nitty Gritty • Start up!

Abbildung 3.4 zeigt den Vorgang in der UML-Darstellung (Sequenzdiagramm). Der linke Pfeil gibt die Zeitachse an, die senkrechten Balken die Lebensdauer von Objekten. Die Löschung des Konto-Objektes ist mit einem Kreuz gekennzeichnet, das Männchen symbolisiert den Anwender.

Danach kann man auf das Objekt zugreifen:

```
int kontonummer = kundenkonto.nummer;
kundenkonto.einzahlen(100.0);
```

Bedeutung von Klassen in Java

Java ist als objektorientierte Sprache komplett klassenbasiert. Anders als in C++ gibt es keine globalen Funktionen, Variablen oder Konstanten. Insbesondere gibt es auch keine globale `main`-Funktion. Die Ausführung einer Applikation beginnt mit der `main`-Methode derjenigen Klasse, die als Parameter an den Interpreter übergeben wurde. Da jede Klasse ihre eigene `main`-Methode enthalten kann, könnte zum Beispiel eine Gruppe mehrerer Klassen verschiedene Applikationen darstellen, je nachdem, mit welcher Klasse gestartet wurde. Auch Applets basieren auf Klassen. Wie wir später sehen werden, ist jedes Applet eine Unterklasse der Klasse *java.applet.Applet*.

Spezialitäten von Variablen und Methoden

Neben Instanzvariablen gibt es manchmal noch Felder, die der Klasse als Gesamtheit zugeordnet sind, also von allen Instanzen gemeinsam genutzt werden. Diese heißen *Klassenvariablen*. Analog existieren auch *Klassenmethoden*, die unabhängig von Objekten ausgeführt werden können. Ein Beispiel ist die bereits erwähnte `main`-Methode.

Überladen von Methoden

Methodenüberladung (Overloading) heißt, dass mehrere Methoden denselben Namen haben können. Welche von den gleichnamigen Methoden aufgerufen wird, hängt dann von den Aufrufparametern zur Laufzeit ab. Durch Methodenüberladung können Gruppen von semantisch gleichwertigen Methoden zusammengefasst werden. Für den Programmierer ist es sicherlich übersichtlicher, wenn alle Methoden, die gleichwertige Aufgaben realisieren, auch den gleichen Namen haben.

```
void ueberweisen(Konto zielkonto, double betrag) ...
void ueberweisen(int zielkontoNummer, double betrag) ...
```

3.2 Vererbung

Neben der *Kapselung*, also der Zusammenfassung von Daten und Funktionen in Klassen, ist die Vererbung das wichtigste objektorientierte Prinzip. Objekte können Eigenschaften von anderen Objekten bzw. deren Klassen »erben«, d. h. sie übernehmen deren Eigenschaften, sofern sie nicht in der eigenen Definition anders definiert sind. Damit muss man identischen Code nur einmal programmieren, was auch aus Wartbarkeitsgründen vorteilhaft ist.

3.2.1 Super- und Subklassen

Ausgehend von einer *Superklasse* (häufig auch: *Oberklasse* oder *Basisklasse*) können weitere Klassen abgeleitet werden, welche das Verhalten der Superklasse erben. Diesem Verhalten werden in der *Subklasse* (oder Unterklasse) zusätzliche Variablen und Methoden zugefügt, die Subklasse wird spezialisiert.

Beispiel

Vertiefen wir unser Konto-Beispiel. Neben dem allgemeinen Konto gebe es zwei abgeleitete Kontenarten:

- ein Sparkonto, für das zusätzlich Zinsen anfallen
- ein Girokonto, das überzogen werden darf

Superklasse

Die Eigenschaften (Konto-)*nummer* und *stand* sind bekannterweise nicht nur Eigenschaften von Sparkonten, sondern von allen Kontoarten. Daher werden diese Eigenschaften der Superklasse *Konto* zugeordnet. Diese Klasse enthält sämtliche Eigenschaften, die für alle Konten gelten.

Subklasse

Die Eigenschaft *kreditlimit* ist eine spezielle Eigenschaft von Girokonten. Die Klasse *Girokonto* wird von der Superklasse *Konto* abgeleitet und erbt sämtliche Eigenschaften. In der Subklasse *Girokonto* wer-

den nur noch die Eigenschaften hinzugefügt, die diese Klasse weiter spezialisieren. Auf diese Weise müssen allgemeine Eigenschaften nicht für jeden Kontentyp (z. B. Sparkonto, Kreditkartenkonto) neu definiert werden.

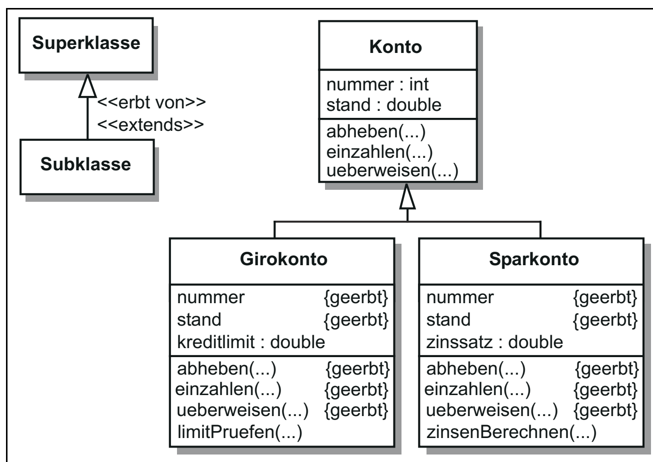

Bild 3.5: Einfache Vererbung

In der *UML* wird Vererbung durch einen Pfeil mit nicht ausgefüllter Pfeilspitze dargestellt, der auf die Superklasse zeigt (siehe Abbildung 3.5).

Vererbung in Java

In Java wird obiges Beispiel folgendermaßen implementiert:

Beispiel

```
class Konto
{
   int     nummer;
   double stand;
   void abheben(double betrag)
   {
      if (stand > betrag)
         stand = stand - betrag;
```

```
    }
    void einzahlen(double betrag)
    {
       stand = stand + betrag;
    }
    void ueberweisen(Konto zielkonto, double betrag)
    {
       abheben(betrag);
       zielkonto.einzahlen(betrag);
    }
}
class Girokonto extends Konto
{
    double kreditlimit;
    boolean limitPruefen(double betrag)
    {
       if (stand - betrag > -kreditlimit)
          return true;
       else
          return false;
    }
}
```

3.2.2 Überschreiben von Methoden

Die Methoden von Subklassen können Methoden ihrer Superklasse *überschreiben*. Sie deklarieren diese in der Subklasse einfach mit dem gleichen Namen und derselben Signatur. Häufig wird in einer überschreibenden Methode zunächst die verschattete Methode der Superklasse aufgerufen und dann eigene Funktionalität ergänzt.

Beispiel

Unser Girokonto hatte bisher noch ein Problem: Man konnte zwar abheben, aber nur so lange noch Geld vorhanden war, das Kreditlimit wurde ignoriert. Man könnte nun zwar eine eigene Methode *abhebenMitLimit()* schreiben, aber dank Überschreiben ist es noch einfacher:

```
class Girokonto extends Konto
{
   [...]
   void abheben(double betrag)
   {
      if (limitPruefen(betrag))
         stand = stand - betrag;
   }
}
```

Wenn wir mit Konten arbeiten, erkennt die Java-Laufzeitumgebung automatisch, dass es sich um ein Girokonto handelt (also den spezielleren Typ), und ruft immer die korrekte Methode auf. Man spricht auch von *virtuellen Methoden*.

In obigem Code-Beispiel habe ich denjenigen Teil weggelassen, der für den aktuell erläuterten Sachverhalt irrelevant ist. Eine Auslassung im Code ist mit [...] *gekennzeichnet.*

3.2.3 Mehrfachvererbung und Interfaces

Einfachvererbung

Mehrfachvererbung bedeutet, dass eine Klasse von mehreren Superklassen abgeleitet ist. Dies kann jedoch zu Mehrdeutigkeiten führen, wenn in den Superklassen ein und desselben Objektes ein Attribut oder eine Methode gleichen Namens vorkommt. Außerdem kann man sich in einem solchen Geflecht von Vererbungsstrukturen leicht verirren.

Deshalb unterstützt Java nur Einfachvererbung, das heiß jede Klasse (bis auf die Urklasse *Object*) besitzt genau eine Superklasse.

Interfaces

Trotzdem möchte man häufig Klassen definieren, die die Schnittstellen mehrerer Basisklassen erfüllen. Beispielsweise könnte man eine »Klasse« *Verzinsbar* geschrieben haben, die gemeinsame Schnittstellen für Wertpapiere und Sparkonten anbietet. Von einer solchen Klasse können wir aber in unserem Beispiel nicht mehr ableiten, da das *Sparkonto* bereits von *Konto* abgeleitet ist.

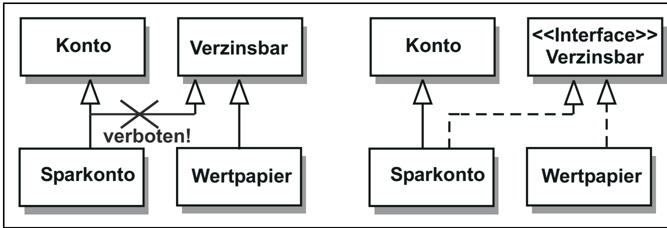

Bild 3.6: Interfaces

Zu diesem Zweck gibt es in Java *Interfaces*. Dies sind rein abstrakte Klassen ohne Instanzvariablen und Methodenimplementierungen. Das heißt, die Methoden werden an dieser Stelle nicht implementiert, sondern es wird nur die Signatur (der Methodenkopf mit der Parameterliste) angegeben. Damit ist eine Mehrdeutigkeit ausgeschlossen, man kann aber damit das Vorhandensein von Methoden garantieren. Wenn eine Klasse ein Interface erfüllt, spricht man vom Implementieren. Die Klasse muss dann all jene Methoden implementieren, deren Deklaration im Interface enthalten ist.

Beispiel

```
interface Verzinsbar
{
   double zinsenBerechnen();
}
class Sparkonto extends Konto implements Verzinsbar
{
   [...]
   double zinsenBerechnen()
   {
      [...]
   }
}
```

3.2.4 Vorteile/Nachteile der Vererbung

Der Einsatz von Vererbung bringt einige wichtige Vorteile:

Wiederverwendbarkeit

Das Verhalten einer Superklasse wird an Subklassen vererbt. Somit wird der Programmcode der Superklasse auch in den Subklassen verwendet und muss daher nicht erneut geschrieben werden. Dadurch entfällt auch die parallele Wartung (und Änderung) mehrerer ähnlicher Codesequenzen für alle Subklassen – eine unangenehme Konstellation, die eine riskante Fehlerquelle darstellt.

Konsistenz von Schnittstellen

Methoden und Variablen, die bereits in der Superklasse definiert wurden, sind auch für sämtliche Subklassen gleich. Damit sind einheitliche Schnittstellen wenigstens bezüglich des in der Superklasse definierten Verhaltens gewährleistet. Diese Schnittstellen sind so auch in allen Subklassen gleich.

Rapid Prototyping

Durch Benutzung von wiederverwendbaren Komponenten, welche durch Klassenhierarchien definiert werden können, kann der Entwicklungsaufwand für neue Projekte auf das Erstellen der projektspezifischen neuen Komponenten beschränkt werden. Der Ansatz des »Rapid Prototyping« besteht darin, möglichst schnell einen Grundstock von Softwarekomponenten zum Laufen zu bringen und dann Schritt für Schritt weitere Komponenten hinzuzufügen.

Information Hiding

Der Anwender der Klassenhierarchien muss nur die öffentlichen Schnittstellen der Klassenhierarchien verstehen. Die konkrete Implementierung bleibt ihm dabei verborgen. Mit Hilfe verschiedener Zugriffsrechte kann die Sichtbarkeit der Klassenbestandteile innerhalb der Vererbungshierarchie fein gesteuert werden.

In manchen Fällen kann sich der Einsatz von Vererbung auch als Nachteil herausstellen:

Effizienz

Allgemein gehaltene Methoden in Superklassen sind oft langsamer und weniger effizient als für den Einzelfall »maßgeschneiderte« Methoden. Hier hat eine elegantere Programmierung und bessere Les-

barkeit des Programmcodes Vorrang vor höchstoptimierten, jedoch nicht wiederverwendbaren Lösungen.

Komplexität

Das Vererbungskonzept kann jedoch im Extremfall auch zum genauen Gegenteil führen. Unendlich verzweigte und in die Tiefe gehende Klassen- und Interface-Hierarchien sind schwer zu durchschauen und machen den Programmcode wenig transparent. Hier muss an die Erfahrung und das Fingerspitzengefühl des Programmierers appelliert werden. Nicht selten geschieht es, gerade in den Anfängen der Java-Programmierung, dass die Begeisterung über ein elegantes Vererbungskonzept den Blick für einfache und durchschaubare Klassenhierarchien trübt. Vor jeder Definition einer Klassen- und Interface-Hierarchie sollten Sie sich daher genau überlegen, ob zur Lösung dieses Problems wirklich eine Klassenhierarchie nötig ist oder ob dieses Problem auch mit einfacheren Konstrukten zu lösen ist.

Einstein-Zitat

Nicht umsonst zitiert der Erfinder der Programmiersprache C++ (von dessen Vererbungskonzept das von Java abstammt), Bjarne Stroustrup, bezüglich des Designs von C++-Klassen den Mathematiker Albert Einstein:

> *»Haltet die Dinge einfach: So einfach wie möglich, aber nicht einfacher.«*