

Einführung

Mit Klassen, Objekten und ihren Elementen lernten wir in der letzten Kurseinheit wichtige Elemente der objektorientierten Programmierung (in Java) kennen. Im Folgenden werden wir uns nun erneut mit dem zentralen Konzept der **Generalisierung und Spezialisierung** und seiner Umsetzung in Java beschäftigen. Dazu gehört zunächst die **Vererbung** zwischen Klassen. Durch die Vererbung entsteht eine **ist-ein-Beziehung** zwischen den Objekten der **Unterklasse** und denen der **Oberklasse**. Wir werden auch noch weitere wichtige Themen im Bereich der Vererbung behandeln, unter anderem das **Substitutionsprinzip**, **Polymorphie** sowie **abstrakte Klassen** und **Schnittstellen**.

Um ungewollten Zugriff auf Klassenelemente zu vermeiden und um das **Geheimnisprinzip** umzusetzen, kann in Java der **Zugriff** auf Klassen und ihre Elemente mit Hilfe von **Zugriffsmodifikatoren** kontrolliert werden. Diese Kontrolle wird durch die Verwendung von **Paketen**, die in großen Projekten helfen, die Klassen zu organisieren, unterstützt.

Am Ende der Kurseinheit werden wir Klassen zur Darstellung und Verarbeitung von **Zeichenketten** genauer betrachten.

Lernziele

- Das Konzept der Vererbung und seine Umsetzung in Java, einschließlich der Begriffe Oberklasse, Unterklasse, abstrakte Klasse, Schnittstelle, Überschreiben, Verdecken, Polymorphie und Substitutionsprinzip erklären und angemessen verwenden können.
- Den statischen und dynamischen Typ von Ausdrücken bestimmen können.
- Den Unterschied zwischen dynamischer und statischer Bindung erläutern können und wissen, welche Bindung in Java bei welchen Elementen verwendet wird.
- Die Klasse `Object` und ihre Rolle in Java kennen.
- Wissen, was bei der Erzeugung von neuen Objekten geschieht.
- Die Bedeutung der Schlüsselwörter `this` und `super` kennen und sie verwenden können.
- Java-Programme in Paketen organisieren und Klassen aus Paketen nutzen können.
- Das Geheimnisprinzip erklären können und Sprachkonstrukte kennen, die die Umsetzung in Java unterstützen.
- Zugriffskontrolle verstehen, die verschiedenen Zugriffsmodifikatoren und ihre Auswirkungen auf die Sichtbarkeit von Klassen und ihren Elementen kennen.
- Die Darstellung der verschiedenen Elemente, wie zum Beispiel Pakete, Schnittstellen, abstrakte Klassen und Sichtbarkeiten, in der UML kennen.
- Wissen, wie Zeichenketten in Java repräsentiert werden und mit welchen Klassen sie am besten bearbeitet werden können, und diese Klassen verwenden.

22 Vererbung und Klassenhierarchien

Bisher war unsere Diskussion auf Klassen beschränkt, bei denen die Attribute und Methoden der Klassen immer von Grund auf neu gebildet wurden. Beziehung zwischen Klassen gab es bisher nur in Form von Assoziationen.

In der ersten Kurseinheit lernten wir das Konzept der Generalisierung und Spezialisierung kennen. In diesem Kapitel wollen wir dieses Konzept und seine Umsetzung in Java näher kennen lernen. Durch die Verwendung von Generalisierung und Spezialisierung entstehen Taxonomien. Taxonomien kennen wir aus der Biologie: Eine Taxonomie ist dort eine biologische Systematik, die dazu dient, Lebewesen gemäß ihrer angenommenen und nachgewiesenen Verwandtschaftsbeziehung zu klassifizieren. Dabei werden bestimmte Merkmale herangezogen. Eine Taxonomie der Reptilien (oder Kriechtiere) zeigt ausschnittsweise Abb. 22-1. Alle Kriechtiere einer Klasse, beispielsweise alle Schuppentiere, weisen die gleichen Eigenschaften auf. Den Kriechtieren nachgeordnete Klassen, wie die Schlangen, verfügen ebenfalls über die Eigenschaften der Schuppentiere, sind aber zugleich spezialisierte Schuppentiere mit besonderen Merkmalen.

Taxonomie

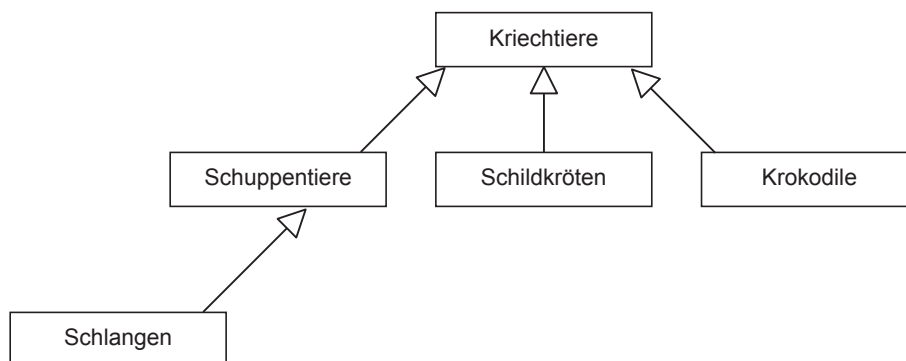


Abb. 22-1: Taxonomie der Kriechtiere

Die Taxonomien der Biologie hatten Vorbildfunktion für die systematische Betrachtung anderer wissenschaftlicher Disziplinen. Heute wird der Begriff Taxonomie als Synonym für alle Arten von Klassifikationsschemata verwendet. Auch die Gegenstände und Begriffe, mit denen wir es bei der Programmentwicklung zu tun haben, können wir hierarchisch strukturieren, und wir verwenden Begriffe aus der Biologie, wie Vererbung, Spezialisierung oder Generalisierung, um die Beziehungen zwischen den Klassen einer Klassenhierarchie zu charakterisieren.

22.1 Vererbung

Bei der Klassenvererbung (engl. *class inheritance*) erbt eine Unterklasse *U* die Attribute und Methoden einschließlich ihrer Implementierung von der Oberklasse *O*, von der *U* abgeleitet wurde. Mittels Vererbung erhalten wir die Möglichkeit, auszudrücken, dass eine Klasse (hier *U*) ähnlich einer anderen Klasse (hier *O*) ist, mit

Klassenvererbung

Unterklasse

Oberklasse

ist-ein-Beziehung

gewissen Unterschieden, die sie spezieller machen. Um diese Unterschiede auszudrücken, können der abgeleiteten Klasse neue Attribute und Methoden hinzugefügt werden, oder die Implementierungen der geerbten Methoden verändert werden. Durch die Vererbung wird zwischen Unter- und Oberklasse eine ist-ein-Beziehung erzeugt.

Betrachten wir noch einmal eine mögliche Klassenhierarchie für die Artikel eines Blumenladens (Abb. 22.1-1). Beispielsweise ist jede Pflanze ein Artikel, umgekehrt gilt dies jedoch nicht.

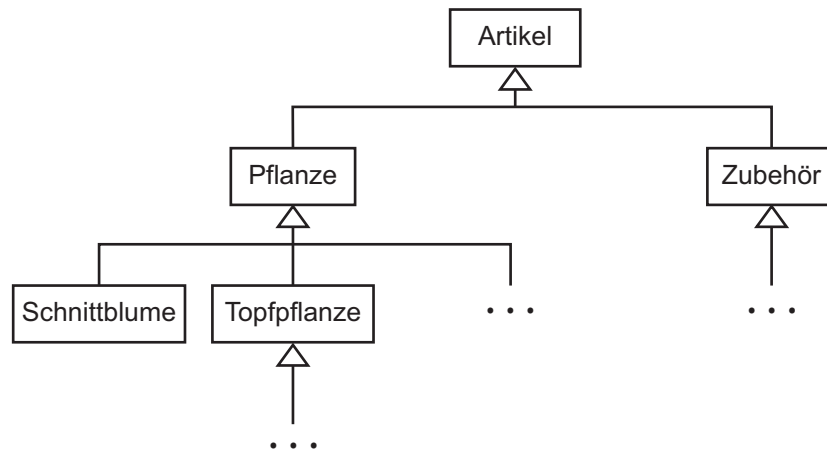


Abb. 22.1-1: Klassenhierarchie für die Artikel eines Blumenladens

22.2 Vererbung in Java

Wollen wir nun in Java ausdrücken, dass die Klasse `Pflanze` eine Unterklasse der Klasse `Artikel` ist, so geschieht dies mit Hilfe des Schlüsselworts `extends` und dem Namen der Oberklasse. [JLS: § 8.1.4]

```

class Artikel {
    double preis;
    // ...

    double lieferePreis() {
        return this.preis;
    }

    void legePreisFest(final double neuerPreis) {
        this.preis = neuerPreis;
    }

    // ...
}

class Pflanze extends Artikel {
}
  
```

Wenn nun die Klasse `Artikel` das Attribut `preis` und die zugehörigen Getter- und Setter-Methoden `lieferePreis()` und `legePreisFest()` besitzt, so besitzt die Klasse `Pflanze` diese ebenso. Wir können folglich an einem Objekt der Klasse `Pflanze` auf die Attribute und Methoden der Oberklasse zugreifen.

```
Pflanze p = new Pflanze();
p.legePreisFest(20.0);
double preis = p.preis; // preis hat den Wert 20.0
preis = p.lieferePreis(); // preis hat auch den Wert 20.0
```

Eine Unterklasse kann eigene Attribute oder Methoden hinzufügen. Wir ergänzen die Klasse `Pflanze` um das Attribut `lagertemperatur`, das angibt, bei welcher Temperatur die Pflanze gelagert werden muss, und um die zugehörigen Getter- und Setter-Methoden.

```
class Pflanze extends Artikel {
    double lagertemperatur;
    // ...

    double liefereLagertemperatur() {
        return this.lagertemperatur;
    }

    void legeLagertemperaturFest(final double temp) {
        this.lagertemperatur = temp;
    }

    // ...
}
```

Auf diese Attribute und Methoden kann nur an Objekten der Klasse `Pflanze` zugegriffen werden, und nicht an Objekten der Klasse `Artikel`.

```
Pflanze p = new Pflanze();
p.legePreisFest(10.0);
p.legeLagertemperaturFest(21.0);
Artikel a = new Artikel();
a.legePreisFest(10.0); // ist möglich
a.legeLagertemperaturFest(15.0); // ist nicht möglich, da die
                                // Klasse Artikel diese
                                // Methode nicht besitzt.
```

Jede Klasse definiert einen eigenen Objekttyp. Durch die Vererbung entstehen zwischen den Objekttypen Beziehungen. Ein durch eine Klasse `U` festgelegter Typ ist genau dann ein Subtyp einer durch die Klasse `O` festgelegten Typs, wenn `U` eine Unterklasse von `O` ist oder `U` und `O` identisch sind. Wir sagen auch, der Objekttyp `U` ist verträglich mit dem Objekttyp `O`. Der Typ des `null`-Verweises ist mit allen Objekttypen verträglich. [JLS: § 4.10.2]

Subtyp

Typverträglichkeit

finale Klasse

Bisweilen ist es nicht erwünscht, dass von einer Klasse Subtypen gebildet werden. In einem solchen Fall kann eine Klasse mit Hilfe des Schlüsselwortes `final` als `final` deklariert werden [JLS: § 8.1.1.2]. So kann verhindert werden, dass Unterklassen von `DiverseDekoration` gebildet werden:

```
class Zubehoer extends Artikel {
    // ...
}

final class DiverseDekoration extends Zubehoer {
    // ...
}
```

Selbsttestaufgabe 22.2-1:

Gegeben sei die folgende Klassenhierarchie:

```
class Person {
    String name;

    public void gebeNameAus() {
        System.out.println(name);
    }
}

class Student extends Person {
    long matrikelnr;

    public void gebeMatrikelnrAus() {
        System.out.println(matrikelnr);
    }
}

class Mitarbeiter extends Person {
    int raumnr;
    Organisationseinheit abteilung;

    public void gebeRaumnummerAus() {
        System.out.println(raumnr);
    }
}

class Professor extends Mitarbeiter {
}

class Organisationseinheit {
    String name;
}

class Lehrgebiet extends Organisationseinheit {
    Professor inhaber;
}
```

Welche der folgenden Attributzugriffe und Methodenaufrufe erzeugen Übersetzungsfehler und warum?

```
new Student().gebeMatrikelnrAus()
new Mitarbeiter().abteilung.inhaber
new Person().gebeNameAus()
new Professor().gebeRaumnummerAus()
new Student().gebeNameAus()
new Professor().abteilung.name
new Mitarbeiter().gebeMatrikelnrAus()
new Lehrgebiet().inhaber.gebeRaumnummerAus()
new Person().getRaumnummerAus()
```



Selbsttestaufgabe 22.2-2:

Geben Sie alle Subtypbeziehungen für die Klassenhierarchie aus Selbsttestaufgabe 22.2-1 an.



Selbsttestaufgabe 22.2-3:

Implementieren Sie die im Folgenden beschriebene Klassenhierarchie in Java, Methoden müssen sie nicht implementieren:

In dem Blumenladen nehmen Personen verschiedene Rollen an. Jede Person hat einen Namen. Über reguläre Kunden gibt es keine weiteren Informationen. Premiumkunden besitzen eine Kundennummer, und ihre Adresse wird gespeichert. Bei Angestellten werden die Adresse, das Gehalt und die Sozialversicherungsnummer gespeichert.



22.3 Substitutionsprinzip

Vererbung bedeutet, dass jedes Objekt der Unterklasse auch eines der Oberklasse ist. Jede Pflanze ist somit auch ein Artikel. Ein Objekt der Unterklasse kann somit überall verwendet werden, wo ein Objekt der Oberklasse erwartet wird. Diesen Umstand nennen wir Substitutionsprinzip. Objekte der Unterklasse substituieren Objekte der Oberklasse. Dies ist immer möglich, da die Unterklasse alle Attribute und Methoden der Oberklasse erbt. Wir können deshalb einer Verweisvariablen vom Typ der Oberklasse ein Objekt der Unterklasse zuweisen:

Substitutionsprinzip

```
Artikel a = new Pflanze();
```

Dabei muss beachtet werden, dass an `a` dennoch nur auf Attribute und Methoden der Klasse `Artikel` zugegriffen werden kann.

```

a.legePreisFest(15.0);           // ist möglich
a.legeLagertemperaturFest(10.0); // ist nicht möglich, da die
                                // Klasse Artikel diese
                                // Methode nicht besitzt

```

`instanceof` Manchmal ist es nötig zu prüfen, ob eine Variable auf ein Objekt einer Unterklasse verweist. Dies kann mit Hilfe des Operators `instanceof` geschehen. Der Ausdruck `x instanceof Y` liefert genau dann `true`, wenn `x` nicht der `null`-Verweis ist und wenn die Klasse des Objekts, auf das `x` verweist ein Subtyp von `Y` ist. [JLS: § 15.20.2]

```

Artikel a = new Artikel();
Artikel p = new Pflanze();
boolean aIstPflanze = a instanceof Pflanze; // liefert false
boolean pIstPflanze = p instanceof Pflanze; // liefert true

```

explizite
Typumwandlung

Wollen wir nun für das Objekt, auf das `p` verweist, die Lagertemperatur erfragen, so wird es nötig, den Typ des Verweises zu ändern. Dies geschieht mit Hilfe einer expliziten Typumwandlung. [JLS: § 5.1.6]

```

// erzeugt einen Übersetzungsfehler:
double temp1 = p.liefereLagertemperatur();
// liefert das gewünschte Ergebnis:
double temp2 = ((Pflanze) p).liefereLagertemperatur();

```

Die explizite Typumwandlung kann auch in Verbindung mit einer Zuweisung verwendet werden.

```

Artikel k = new Pflanze();
Pflanze m = (Pflanze) k;

```

ClassCast-
Exception

Eine explizite Typumwandlung `(A) x` ist genau dann möglich, wenn der Typ des Objekts, auf das `x` verweist, mit `A` verträglich ist. Andernfalls wird zur Laufzeit eine `ClassCastException` auftreten:

```

Artikel b = new Artikel();
double temp = ((Pflanze) b).liefereLagertemperatur();
// erzeugt ClassCastException

```

Bemerkung 22.3-1: Übersetzungsfehler

Sowohl bei einer expliziten Typumwandlung, als auch bei einem `instanceof`-Ausdruck kann ein Übersetzungsfehler auftreten, wenn es zwischen den Typen auf Grund der Typhierarchie auf keinen Fall eine Subtypbeziehung geben kann. ┘

Beispiel 22.3-1:

```

class A {
}

class B extends A {
}

class C extends A {
}

B b = new B();
boolean isInstance = b instanceof C; // liefert Übersetzungsfehler
C c = (C) b; // liefert Übersetzungsfehler

```

Die beiden Anweisungen liefern Übersetzungsfehler, da ein Exemplar vom Typ B auf Grund der Klassenhierarchie niemals ein Exemplar vom Typ C sein kann. ┘

Neben der expliziten Typumwandlung gibt es auch für Verweise eine implizite Typumwandlung [JLS: § 5.1.5]. Der Verweis einer Variablen *x* kann einer Variablen *y* ohne explizite Typumwandlung zugewiesen werden, wenn der Typ von *x* ein Subtyp von *y* ist.

implizite
Typumwandlung

```

Pflanze s = new Pflanze();
Artikel t;
t = s;

```

Auf Grund des Substitutionsprinzip ist es möglich, dass der Typ einer Verweisvariablen nicht identisch mit dem Typ des Objekts ist, auf das sie verweist. Um zwischen diesen beiden Typen zu unterscheiden, führen wir die Begriffe statischer und dynamischer Typ ein.

statischer Typ
dynamischer Typ

Definition 22.3-1: Statischer Typ einer Verweisvariablen

Der statische Typ einer Verweisvariablen ist immer der Typ, der bei der Deklaration vereinbart wurde. Er kann sich nicht im Laufe eines Programms verändern. ┘

Definition 22.3-2: Dynamischer Typ einer Verweisvariablen

Unter dem dynamischen Typ einer Verweisvariablen verstehen wir den Typ des Objekts, auf das die Variable zu diesem Zeitpunkt verweist. Da eine Verweisvariable im Laufe eines Programms auf verschiedene Objekte verweisen kann, kann sich auch der dynamische Typ ändern. Der dynamische Typ ist immer ein Subtyp des statischen Typs. ┘

Man kann nicht nur bei Variablen, sondern auch bei Ausdrücken zwischen statischem und dynamischem Typ unterscheiden. Ein Konstruktoraufruf hat immer den gleichen statischen und dynamischen Typ, und zwar die Klasse, deren Konstruktor gerade aufgerufen wurde. Handelt es sich bei dem Ausdruck um einen Methodenaufruf, so ist der statische Typ der Ergebnistyp der Methode und der dynamische derjenige, den das zurückgegebene Objekt zur Laufzeit besitzt. Der statische und dynamische Typ von `this` ist die umgebende Klasse.

Bemerkung 22.3-2: Auswirkungen von Typumwandlungen (Cast)

Wenn ein Cast auf einen Verweis angewendet wird, so wird lediglich der statische Typ verändert. Der dynamische Typ bleibt unverändert. So ist zum Beispiel der statische Typ des Ausdrucks `((Artikel) new Pflanze())` der Objekttyp `Artikel`, der dynamische Typ hingegen `Pflanze`. ┘

Der dynamische Typ kann immer mit Hilfe von `instanceof` überprüft werden.

Der Übersetzer kann immer nur den statischen Typen von Variablen und Ausdrücken bestimmen und überprüfen, da der dynamische Typ häufig erst zur Laufzeit feststeht.

Das Substitutionsprinzip ist auch bei der Argumentübergabe und bei Ergebnistypen anwendbar. Erwartet eine Methode oder ein Konstruktor ein Argument vom Typ `O`, so kann jedes Objekt, dessen Typ mit `O` verträglich ist, als Argument übergeben werden. So können wir zum Beispiel beim Erzeugen eines neuen Rechnungsposten nicht nur ein `Artikel`-Objekt übergeben, sondern auch ein `Pflanze`-Objekt.

```
Pflanze p = new Pflanze();
Rechnungsposten rp = new Rechnungsposten(10, p);
```

Die Methode `liefereArtikel()` der Klasse `Rechnungsposten` hat als Ergebnistyp den Objekttyp `Artikel`. Auf Grund des Substitutionsprinzips kann die Methode durchaus ein `Pflanze`-Objekt zurück liefern. Da der statische Typ eines Methodenaufrufs immer genau dem deklarierten Ergebnistyp entspricht, können wir an dem von der Methode `liefereArtikel()` zurückgelieferten Objekt ohne explizite Typumwandlung nur Methoden der Klasse `Artikel` aufrufen.

Selbsttestaufgabe 22.3-1:

Bestimmen Sie den statischen und dynamischen Typ von `x`, `y` und `z`:

```
Artikel x = new Pflanze();
Pflanze y = (Pflanze) x;
Artikel z = new Artikel();
```



Selbsttestaufgabe 22.3-2:

Gegeben sei die folgende Klasse:

```

public class X {
    public static Artikel doSomething(Artikel a) {
        if (a == null) {
            return new Pflanze();
        }
        return a;
    }
}

```

Bestimmen Sie den statischen und dynamischen Typ der folgenden Ausdrücke:

```

new Artikel()
X.doSomething(null)
((Artikel) new Pflanze())
X.doSomething(new Artikel())
((Pflanze) X.doSomething(null))

```

**Selbsttestaufgabe 22.3-3:**

Gegeben sei die folgende Klassenhierarchie:

```

class A {
    double h;

    void p(B r) {
        h += r.i;
    }
}

class B extends A {
    double i;
}

class C extends B {
    int j;

    D g(A f) {
        h += f.h;
        return new D();
    }
}

class D extends B {

    void k(B m) {
        h += m.i;
    }
}

```

```
class E extends A {
    int i;
}
```

Welche der folgenden Anweisungen erzeugen Übersetzungsfehler und welche Laufzeitfehler?

```
A s = new A();
C t = (C) s;
new D().k(new E());
new C().g(s).k(null);
A u = new C().g(new D());
double x = ((B) u).i;
```



22.4 Überschreiben und Verdecken

Die Vererbung von Attributen und Methoden einer Oberklasse und das Hinzufügen eigener Methoden und Attribute in abgeleiteten Klassen ist oft nicht hinreichend, um die für das gegebene Anwendungsproblem notwendige Lösung zu erhalten. So wollen wir beispielsweise, dass alle Artikel eine Methode `gebeInformationenAus()` besitzen, die alle relevanten Informationen über diesen Artikel auf dem Bildschirm ausgibt. Zwar haben alle Klassen zum Beispiel einen Preis, einen Namen und eine Artikelnummer, aber Pflanzen haben zusätzlich noch eine Lagertemperatur. Das bedeutet, dass wir das Verhalten der Methode für die Klasse `Pflanze` anpassen müssen.

Um solche Anforderungen zu bewältigen, ermöglicht es uns Java, ererbte Methoden einer Oberklasse zu überschreiben, um die Implementierung einer geerbten Methode an die Bedürfnisse der Unterklasse anzupassen. Durch Überschreiben wird das Verhalten der Methode in der Unterklasse verändert.

Definition 22.4-1: Überschreiben von Methoden

Überschreiben

*Eine Unterklasse U kann eine geerbte Methode $m()$ durch Definition einer neuen Methode $m()$, die die gleiche Signatur hat wie die geerbte Methode, überschreiben (engl. *override*). Der Ergebnistyp in der Unterklasse muss identisch oder verträglich mit dem in der Oberklasse deklarierten Ergebnistyp sein. [JLS: § 8.4.8.1, § 8.4.8.3]*

Wird an einem Objekt der Unterklasse eine überschriebene Methode aufgerufen, so wird in der Unterklasse implementierte Methode ausgeführt.

Statische Methoden können nicht überschrieben werden. [JLS: § 8.4.8.2]

finale Methode

Eine Methode kann nicht überschrieben werden, wenn sie als `final` vereinbart ist. [JLS: § 8.4.3.3]



Ergänzen wir zuerst die Klasse `Artikel` um die Attribute `artikelnr` und `name`, die zugehörigen Getter- und Setter-Methoden sowie um die Methode `gebeInformationenAus()`.

```
class Artikel {
    double preis;
    String name;
    int artikelnr;

    double lieferePreis() {
        return this.preis;
    }

    void legePreisFest(final double neuerPreis) {
        this.preis = neuerPreis;
    }

    String liefereName() {
        return this.name;
    }

    void legeNameFest(final String neuerName) {
        this.name = neuerName;
    }

    int liefereArtikelnummer() {
        return this.artikelnr;
    }

    void legeArtikelnummerFest(final int neueNr) {
        this.artikelnr = neueNr;
    }

    void gebeInformationenAus() {
        System.out.print("Artikel ");
        System.out.println(this.liefereName());
        System.out.print("Nr: ");
        System.out.println(this.liefereArtikelnummer());
        System.out.print("Preis: ");
        System.out.println(this.lieferePreis());
    }
}
```

Um nun in der Klasse `Pflanze` die Methode entsprechend anzupassen, implementieren wir die Methode erneut.

```
class Pflanze extends Artikel {
    double lagertemperatur;

    double liefereLagertemperatur() {
        return this.lagertemperatur;
    }
}
```

```

void legeLagertemperaturFest(final double temp) {
    this.lagertemperatur = temp;
}

void gebeInformationenAus() {
    System.out.print("Artikel ");
    System.out.println(this.liefereName());
    System.out.print("Nr: ");
    System.out.println(this.liefereArtikelnummer());
    System.out.print("Preis: ");
    System.out.println(this.lieferePreis());
    System.out.print("Lagertemperatur: ");
    System.out.println(this.liefereLagertemperatur());
}
}

```

Wir stellen fest, dass wir auf alle Attribute, sowohl die selbst definierten als auch die geerbten, mit Hilfe von `this` zugreifen können.

Wir haben gelernt, dass Vererbung die Wiederverwendung erleichtert und doppelten Quelltext vermeidet. Jedoch stimmen einige der Anweisungen in der Methode `gebeInformationenAus()` der Klasse `Pflanze` mit denen der Klasse `Artikel` überein. Besser wäre es, wenn wir auf das in der Oberklasse `Artikel` implementierte Verhalten zugreifen können. Dies geschieht mit Hilfe des Schlüsselwortes `super`.

Das Schlüsselwort `super` ermöglicht den Zugriff auf überschriebene Methoden der Oberklasse. So können wir in jeder Methode der Unterklasse mit Hilfe von `super.methodenName()` auf die Implementierung der Oberklasse zugreifen [JLS: § 8.4.8.1, § 15.12.4.9]. Dadurch vermeiden wir in der Klasse `Pflanze` den doppelten Quelltext.

```

class Pflanze extends Artikel {
    double lagertemperatur;

    double liefereLagertemperatur() {
        return this.lagertemperatur;
    }

    void legeLagertemperaturFest(final double temp) {
        this.lagertemperatur = temp;
    }

    void gebeInformationenAus() {
        super.gebeInformationenAus();
        System.out.print("Lagertemperatur: ");
        System.out.println(this.liefereLagertemperatur());
    }
}

```

Selbsttestaufgabe 22.4-1:

Erläutern Sie den Unterschied zwischen Überschreiben und Überladen von Methoden.

**Selbsttestaufgabe 22.4-2:**

Gegeben sei die folgende Klassenhierarchie.

```
class A {
    int x = 10;

    void f() {
        x++;
        System.out.println(x);
    }

    void g() {
        x += x;
        System.out.println(x);
    }
}

class B extends A {

    void f() {
        x--;
        System.out.println(x);
    }
}
```

Welche Ausgabe würden die folgenden Anweisungen erzeugen?

```
A a = new A();
B b = new B();
a.f();
b.f();
b.g();
a.g();
a.f();
b.f();
```

**Selbsttestaufgabe 22.4-3:**

Enthält der folgende Quelltext Fehler und wenn ja, warum?

```
class Ober {
    Unter a() {
        return new Unter();
    }
}
```

```

        Ober b() {
            return new Unter();
        }
    }

    class Unter extends Ober {
        Ober a() {
            return new UnterUnter();
        }
    }

    class UnterUnter extends Unter {
        UnterUnter b() {
            return new UnterUnter();
        }
    }

```



Bisher haben wir gelernt, Methoden zu überschreiben und auf die Implementierungen in der Oberklasse zuzugreifen. Es kann bei einer Klassenhierarchie auch vorkommen, dass eine Unterklasse ein Attribut definiert, das den gleichen Namen wie ein Ererbtes trägt.

```

class Ober {
    int x;
}

class Unter extends Ober {
    double x;
}

```

Verdecken In diesem Fall verdeckt (engl. *hide*) das Attribut `double x` der Klasse `Unter` das von `Ober` geerbte Attribut `int x`. [JLS: § 8.3.3.2]

Will man nun explizit auf `int x` zugreifen, so ist dies mit Hilfe von `super` möglich. [JLS: § 15.11.2]

```

class Ober {
    int x;
}

class Unter extends Ober {
    double x;

    void test() {
        int a = super.x; // Zugriff auf int x aus Ober
        double b = x;    // Zugriff auf double x aus Unter
    }
}

```


Bemerkung 22.4-1: Keine Aneinanderreihung des Schlüsselworts `super`

Es ist nicht zulässig, das Schlüsselwort `super` wie in

```
super.super.gebeInformationenAus();
```

mehrfach aneinander zu reihen, um auf diese Weise über mehr als eine Stufe in der Klassenhierarchie hinweg auf die Implementierung einer Methode oder ein verdecktes Attribut zuzugreifen.

┘

Selbsttestaufgabe 22.4-4:

Gegeben sei die folgende Klassenhierarchie:

```
class A {
    int x = 10;

    void f(int x) {
        System.out.println(x);
        System.out.println(this.x);
    }
}

class B extends A {
    int x = 2;

    void f() {
        int x = 3;
        System.out.println(super.x);
        System.out.println(x);
        System.out.println(this.x);
        super.f(x);
    }

    void f(int x) {
        System.out.println(this.x);
    }
}
```

Welche Ausgabe erzeugen die folgenden Anweisungen?

```
new A().f(10);
new B().f();
new B().f(2);
```

◇

22.5 Die Klasse Object

Alle Klassen von Java sind implizit oder explizit Unterklassen der vordefinierten Klasse `Object`. Wenn eine Klassendefinition keinen `extends`-Ausdruck enthält, fügt der Java-Übersetzer implizit den `extends`-Ausdruck für die Klasse `Object` hinzu. [JLS: § 4.3.2]

Damit steht die Klasse `Object` in jeder Klassenhierarchie an der Wurzel. Somit besitzt jede Java-Klasse die Methoden der Klasse `Object`. Bei einigen Methoden ist es jedoch sinnvoll, wenn eine Klasse diese überschreibt.

Lassen Sie uns einige Methoden der Klasse `Object` betrachten.

- `equals()` ist eine Methode, die einen Wahrheitswert zurück liefert und überprüft, ob das Objekt, an das die `equals()`-Methode gerichtet ist, dem im Argument der Methode referenzierten Objekt gleich ist. In der Implementierung der Klasse `Object` liefert die Methode `equals()` genau dann `true`, wenn Objekt-Identität vorliegt. Diese Semantik ist identisch mit dem Vergleich zweier Objektverweise mit Hilfe des „`==`“-Operators. So wird, falls `a` und `b` Objekte der Klasse `Object` sind,

```
a.equals(b)
```

das gleiche Ergebnis liefern, wie

```
a == b
```

In der Praxis wird diese vordefinierte Bedeutung der `equals()`-Methode oft als zu einschränkend und wenig nützlich empfunden. Deswegen wird `equals()` in Unterklassen von `Object` häufig überschrieben, um Objekte z. B. auf die Gleichheit relevanter Attribute hin zu testen.

- `toString()` liefert eine Zeichenkettenrepräsentation des Objekts, also ein Objekt der Klasse `String`, die dann ausgegeben oder dauerhaft auf einem Speichermedium abgelegt werden kann. Diese Methode sollte in der Regel von anderen Klassen überschrieben werden, so dass nützliche Informationen über das Objekt in Form einer Zeichenkette zurückgeliefert werden.

Eine weiterer Vorteil des Überschreibens der `toString()`-Methode ist deren indirekte Verwendung in den Methoden `print()` und `println()` der Klasse `PrintStream`, die bei jeder unserer Ausgaben der Form `System.out.print()` zum Einsatz kommen. Bekommt eine solche Methode ein Objekt übergeben, so wird automatisch das Ergebnis der `toString()`-Methode ausgegeben. Würde beispielsweise die `toString()`-Methode der Klasse `X` immer `"XXX"` zurück liefern, so würde die Anweisung `System.out.println(new X())` die Zeichenkette `"XXX"` ausgeben, ohne dass explizit die `toString()`-Methode aufgerufen wird.

- `getClass()` gibt ein Objekt der vordefinierten Klasse `Class` zurück. Ein `Class`-Objekt enthält detaillierte Informationen über die Klasse, deren Exemplar das Objekt ist, das den Methodenaufruf `getClass()` empfangen hat.

Bemerkung 22.5-1: Überschreiben von Methoden

Bevor Sie in einer eigenen Klasse eine Methode wie z.B. `equals()` oder `toString()` überschreiben, sollten Sie in jedem Fall die Dokumentation der Methode studieren.

└

Die Klasse `Object` unterstützt noch viele weitere Methoden, die wir im Rahmen dieses Kurses nicht besprechen.

22.6 Konstruktoren und Erzeugung von Objekten einer Klassenhierarchie

Die Erzeugung von Objekten einer Klassenhierarchie ist ein mehrstufiger Vorgang, den wir in diesem Abschnitt genauer betrachten wollen. Da alle Klassen, die wir bisher in diesem Kapitel als Beispiele verwendet haben, einen Standardkonstruktor besaßen, sind uns die Abläufe bei der Erzeugung von Objekten zunächst verborgen geblieben.

Nun ergänzen wir die Klasse `Artikel` um zwei Konstruktoren.

```
class Artikel {
    double preis;
    String name;
    int artikelnr;

    Artikel(final double preis, final int artikelnr) {
        this.legePreisFest(preis);
        this.legeArtikelnummerFest(artikelnr);
    }

    Artikel(final double preis, final int artikelnr,
            final String name) {
        this(preis, artikelnr);
        this.legeNameFest(name);
    }

    // Methoden
    // ...
}
```

Die Klasse `Artikel` bietet nun keinen Standardkonstruktor mehr an. Würden wir die Klasse `Pflanze` nicht weiter verändern, würde der Übersetzer einen Fehler melden. Denn bei der Ausführung eines Konstruktors während der Erzeugung eines Objekts werden automatisch auch Konstruktoren der Oberklassen aufgerufen. Besitzt nun die direkte Oberklasse, in unserem Fall `Artikel`, keinen Standardkonstruktor mehr, so muss in jedem Konstruktor der Unterklasse, hier `Pflanze`, explizit ein Konstruktor der gleichen Klasse oder der Oberklasse aufgerufen werden [JLS: § 8.8.7]. Wir erinnern uns, dass wir einen Konstruktor der gleichen Klasse mit

Hilfe von `this()` aufrufen können. Ein Konstruktor der Oberklasse kann entsprechend mit Hilfe von `super()` aufgerufen werden.

```
class Pflanze extends Artikel {
    double lagertemperatur;

    Pflanze(final double preis, final int artikelnr,
            final double lagertemp) {
        super(preis, artikelnr);
        this.legeLagertemperaturFest(lagertemp);
    }

    Pflanze(final double preis, final int artikelnr,
            final String name, final double lagertemp) {
        super(preis, artikelnr, name);
        this.legeLagertemperaturFest(lagertemp);
    }

    // Methoden
    // ...
}
```

Bemerkung 22.6-1:

Besitzt die direkte Oberklasse keinen Standardkonstruktor, so muss es sich bei der ersten Anweisung eines Konstruktors der Unterklasse entweder um einen `this()`- oder einen `super()`-Aufruf handeln. Es ist jedoch nur einer der beiden möglich. Besitzt die direkte Oberklasse einen Standardkonstruktor und die erste Anweisung im Konstruktor der Unterklasse ist kein `this()`- oder `super()`-Aufruf, so wird implizit ein `super()`-Aufruf für den Standardkonstruktor eingefügt. ┘

Wenn nun ein Konstruktor der Klasse `U` aufgerufen wird, reserviert das Java-Laufzeitsystem zunächst Speicher für alle Attribute von `U` und für die aller Oberklassen von `U`. Die allgemeinste Klasse in einer Hierarchie ist immer die vordefinierte Klasse `Object`, da sie immer implizit einer benutzerdefinierten Klassenhierarchie hinzugefügt wird.

Konstruktorausführung Anschließend wird der aufgerufene Konstruktor von `U` mit dem folgenden Verfahren ausgeführt [JLS: § 12.5]:

- Ist die erste Anweisung ein Aufruf eines Konstruktors der gleichen Klasse unter Verwendung von `this` oder ein Aufruf eines Konstruktor der Oberklasse unter Verwendung von `super`, dann wird dieser wiederum nach diesem Verfahren ausgeführt.
- Ist kein expliziter Konstruktoraufruf vorhanden und es handelt sich bei der aktuellen Klasse nicht um `Object`, so wird der Standardkonstruktor der Oberklasse aufgerufen. Ist in diesem Fall in der Oberklasse kein Standardkonstruktor vorhanden, führt dies schon bei der Übersetzung zu einem Fehler.
- Initialisiere die Attribute dieser Klasse.

- Führe den (restlichen) Rumpf des Konstruktors aus.

Abb. 22.6-1 veranschaulicht diese beiden gegenläufigen Prozesse für unsere Klassenhierarchie aus `Artikel` und `Pflanze`.

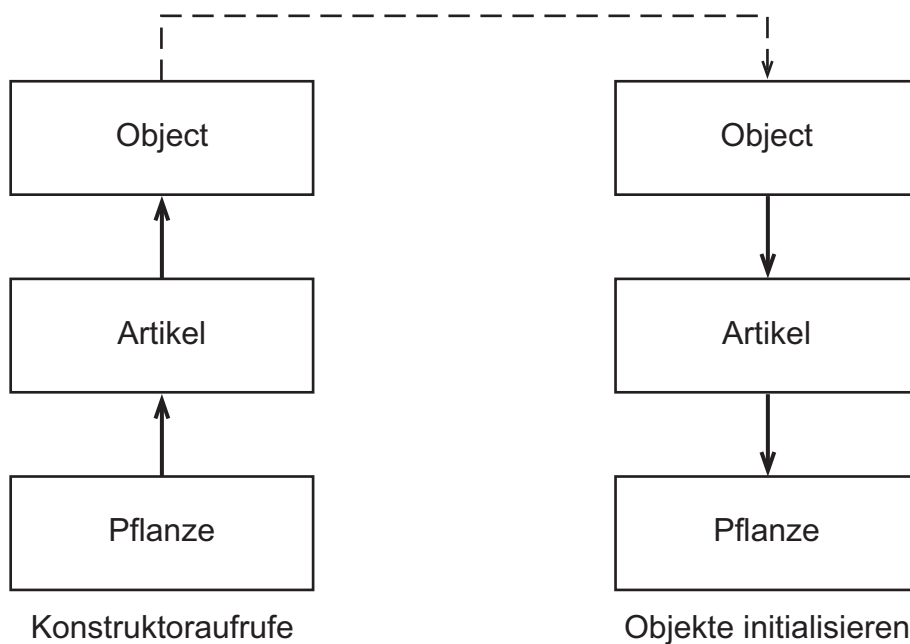


Abb. 22.6-1: Konstruktoraufrufe und Objektbelegung

Bemerkung 22.6-2: Keine Aneinanderreihung des Schlüsselworts `super`

Auch bei Konstruktoraufrufen ist es nicht zulässig, das Schlüsselwort `super` wie in

```
super.super()
```

mehrfach aneinander zu reihen, um auf diese Weise über mehr als eine Stufe in der Klassenhierarchie hinweg einen Konstruktor einer Oberklasse aufzurufen. ┘

22.7 Polymorphie, dynamisches und statisches Binden

Bisher sprachen wir noch nicht darüber, wie entschieden wird, welche Implementierung bei einem Methodenaufruf, insbesondere bei überschriebenen und überladenen Methoden, ausgeführt wird. Es liegt nahe zu vermuten, dass diese Entscheidung bei der Übersetzung getroffen wird. Die Festlegung, welcher Aufruf mit welcher Implementierung verknüpft ist, wird auch als Bindung bezeichnet. Wird diese Bindung während der Übersetzung festgelegt, so nennt man dies statisches Binden (engl. *static binding*).

statisches Binden

Wir haben jedoch vorhin festgestellt, dass der dynamische Typ einer Variablen oder eines Ausdrucks oftmals erst während der Laufzeit bestimmt werden kann. Somit

dynamisches Binden kann die Bindung von Methoden auch häufig erst zur Laufzeit geschehen. Eine Bindung zur Laufzeit wird auch dynamische Bindung (engl. *dynamic binding*) genannt.

Java unterstützt, wie viele andere objektorientierte Programmiersprachen auch, das dynamische Binden. Die „Java Virtual Machine“ (JVM) führt eine dynamische Bindung für Objektverweise in Ausdrücken wie `a.gebeInformationenAus()` durch. Bei der dynamischen Bindung legt nicht der statische Typ der Variablen `a`, sondern die Klasse des Objekts, auf das die Variable zur Laufzeit verweist, fest, welche Methodenimplementierung ausgeführt wird.

Die Unterklassenbildung ermöglicht es uns, ein Objekt einer Unterklasse zu benutzen, wo auch immer ein Objekt der Oberklasse verlangt ist. In Verbindung mit der dynamischen Bindung von Methodennamen an Methodenimplementierungen unterstützt das Konzept der Substituierbarkeit von Objekten die Möglichkeit, vieltypige oder polymorphe Methoden zu definieren.

Polymorphie Grob gesagt bedeutet Polymorphie, dass eine an ein Objekt gesandte Nachricht der Form `o.m(...)` den Aufruf der Methode `m()` bewirkt, die in der Klasse des Objekts, auf das `o` verweist, oder einer der Oberklassen definiert ist.

Lassen Sie uns den Begriff Polymorphie anhand unserer Artikelhierarchie illustrieren. Wir könnten in unserem Programm beispielsweise ein Feld von Artikeln haben. Hinter den einzelnen Elementen könnten sich zum Beispiel auch Pflanzen verbergen. Wollen wir nun die Informationen aller gespeicherten Artikel ausgeben, so können wir das Feld mit Hilfe einer `for`-Schleife durchlaufen.

```
Artikel[] artikel = {new Artikel(20.0,10),
                    new Pflanze(5.0, 2222, 10),
                    new Artikel(1.0,50)};
for (Artikel a : artikel) {
    a.gebeInformationenAus();
}
```

Es wird jedes Mal die gleiche Methode an der gleichen Variablen aufgerufen, jedoch zeigt sie verschiedene Verhaltensweisen, je nachdem, ob `a` auf ein `Artikel`- oder ein `Pflanze`-Objekt verweist.

In der Praxis führt die Polymorphie zu einem klareren Programmentwurf. Darüber hinaus macht ein effektiver Einsatz von Polymorphie große `switch`- und `if`-Anweisungen überflüssig.

Wenn es also möglich ist, dass ein Objekt `o` einer Unterklasse die Rolle eines Objekts einer seiner Oberklassen einnehmen kann und Methoden in Unterklassen überschrieben werden können, stellt sich nun die Frage, wie für eine Nachricht `o.m(...)` die passende Implementierung der Methode `m()` in den Klassen einer Vererbungshierarchie bestimmt wird.

Die Bestimmung der Methode, die zur Laufzeit ausgeführt wird, läuft in zwei Schritten ab [JLS: § 15.12]. Der erste findet bei der Übersetzung statt und bestimmt

die Signatur s der Methode, die zur Laufzeit ausgeführt werden soll. Zur Laufzeit wird dann bei Exemplarmethoden diejenige Methode ausgeführt, die der dynamische Typ mit Signatur s geerbt hat. Diese kann identisch mit der Methode, die zur Übersetzungszeit gefunden wurde, sein, aber sich auch davon unterscheiden, wenn eine der Klassen zwischen dynamischem und statischem Typ diese Methode überschrieben hat. Es wird immer diejenige Methode mit der Signatur s ausgeführt, die am weitesten unten in der Hierarchie des dynamischen Typs zu finden ist.

Um die passende Methode zu finden, muss der statische Typ des Ausdrucks oder der Variablen bestimmt werden, an dem die Methode aufgerufen wird. Während der Übersetzungszeit wird in der Vererbungshierarchie, vom statischen Typ aufsteigend, diejenige Methode $m()$ gesucht, die am besten passt, also die speziellste Signatur besitzt, und die so weit unten wie möglich in der Hierarchie steht. Die Suche kann beendet werden, sobald eine Methode gefunden wurde, deren Parametertypen mit den statischen Typen der Argumente übereinstimmen. Ansonsten wird die Suche bis zur Klasse `Object` fortgesetzt.

Sollte es sich bei m um eine Klassenmethode handeln, so wird diese gebunden und auch zur Laufzeit ausgeführt, unabhängig vom dynamischen Typ. Bei Klassenmethoden gibt es folglich nur die statische Bindung.

Zur Laufzeit wird im Falle, dass $m()$ eine Exemplarmethode ist, vom dynamischen Typ des Objekts aufsteigend aus, diejenige Methode $n()$ ausgeführt, die die gleiche Signatur wie $m()$ hat, sie also überschreibt, und die zuerst gefunden wird, d. h. die so weit unten wie möglich in der Hierarchie steht.

Bemerkung 22.7-1:

Ein Typ A ist spezieller als Typ B , wenn A ein Subtyp von B ist. Bei mehreren Parametern wird verglichen, welche Methode die meisten spezielleren Typen besitzt. Kann keine eindeutige Entscheidung getroffen werden, so wird ein Übersetzungsfehler verursacht.

┘

Bemerkung 22.7-2:

Ist eines der übergebenen Argumente `null`, so ist der statische Typ mit allen Klassen verträglich, im Zweifelsfalls wird jedoch der speziellste angenommen.

┘

Bemerkung 22.7-3: Mögliche Fehler und Warnungen zur Übersetzungszeit

Es können zur Übersetzungszeit unter anderem in den folgenden Fällen Fehler und Warnungen auftreten:

- 1. In der gesamten Hierarchie des statischen Typs wird keine passende Methode gefunden.*
- 2. Der Rückgabetyt, der zur Übersetzungszeit bestimmt wird, passt nicht zur Verwendung des Aufrufs.*
- 3. Bei der Methode $m()$, die zur Übersetzungszeit gefunden wird, handelt es sich um eine Exemplarmethode, die jedoch an einem Typnamen aufgerufen*

wird. Denn eine Exemplarmethode muss immer an einem Objekt aufgerufen werden.

4. Wenn eine statische Methode an einem Verweis auf ein konkretes Objekt aufgerufen wird, so wird eine Warnung erzeugt.
5. Es kann nicht bestimmt werden, welche Methode auszuführen ist, weil mehrere Methoden gleich gut zu den statischen Typen der Argumente passen. ┘

Bemerkung 22.7-4: Mögliche Fehler zur Laufzeit

Ein sehr häufiger Fehler, der zur Laufzeit auftritt, ist eine `NullPointerException`. Diese tritt bei der Ausführung einer Exemplarmethode auf, wenn der Verweis auf das Objekt, an dem die Methode aufgerufen wird, `null` ist. ┘

Selbsttestaufgabe 22.7-1:

Welche Ausgabe wird durch den Aufruf `x.z("Hallo")` in der Methode `test()` der Klasse `E` bei der folgenden Klassenhierarchie erzeugt? Welche Signatur wird zur Übersetzungszeit gefunden und welche Methode dann zur Laufzeit ausgeführt und warum?

```
public class A {
    public void z(String x) {
        System.out.println("A.z");
    }
}

public class B extends A {
    public void z(Object x) {
        System.out.println("B.z");
    }
}

public class C extends B {
    public void z(String x, String y) {
        System.out.println("C.z");
    }
}

public class D extends C {
    public void z(String x) {
        System.out.println("D.z");
    }
}

public class E extends D {
    public void test() {
        C x = new E();
        x.z("Hallo");
    }
}
```



Selbsttestaufgabe 22.7-2:

Finden Sie heraus warum bei der Ausführung der Methode `ZZZ.test()` die Ausgabe `ZZZYYY` erscheint.

```
public class ZZZ {
    public void a(Object x) {
        System.out.print("ZZZ");
    }

    public static void test() {
        ZZZ z = new YYY();
        z.a("Hallo");
        YYY y = new YYY();
        y.a("Hallo");
    }
}

class YYY extends ZZZ {
    public void a(String x) {
        System.out.print("YYY");
    }
}
```



Die Bindung von Attributen erfolgt wesentlich einfacher als die Bindung von Methoden, da es sich um eine statische Bindung handelt. Es wird nur der statische Typ bei der Auflösung des Attributs berücksichtigt. Bei der Bindung von Attributen wird auch nicht zwischen Klassen- und Exemplarattributen unterschieden.

Beispiel 22.7-1: Statische vs. dynamische Bindung

Der nachfolgende Quelltext soll den Unterschied zwischen dynamischer Bindung bei Exemplarmethoden und der statischen Bindung bei (Exemplar-)Attributen verdeutlichen:

```
public class A {
    public int x = 8;

    public int getX() {
        return x;
    }
}

public class B extends A {
    public int x = 10;

    public int getX() {
        return x;
    }
}
```

```
public void testBinding() {  
    A a = new A();  
    B b = new B();  
    A ab = new B();  
    System.out.print(a.x);           // 8  
    System.out.print(b.x);           // 10  
    System.out.print(ab.x);          // 8  
  
    System.out.print(a.getX());      // 8  
    System.out.print(b.getX());      // 10  
    System.out.print(ab.getX());     // 10  
}  
}
```

Auch wenn x ein Klassenattribut wäre, würde dies nichts an den Ausgaben ändern. ┘

Polymorphie erlaubt es uns, polymorphe oder generische Algorithmen (engl. *generic algorithms*) zu schreiben. Ihre Allgemeingültigkeit gestattet uns, Objekte unterschiedlicher Klassen auf eine einheitliche Art und Weise zu manipulieren, sofern der Algorithmus nur gemeinsame Eigenschaften und Verhaltensweisen der verschiedenen Klassen verwendet.

Es könnte zum Beispiel sein, dass ein Ausdruck aller im Lager enthaltenen Artikel erstellt werden soll, wobei diese aufsteigend nach ihrem Preis sortiert werden sollen. Dabei würden nur auf Eigenschaften, die in der Klasse `Artikel` festgelegt werden, zugegriffen werden. Die Sortierung könnte folglich unabhängig von den Typen der einzelnen Objekte erfolgen.

23 Pakete

Mit Paketen bietet Java einen hilfreichen Mechanismus an, um inhaltlich zusammengehörige Klassen und Schnittstellen in einer benannten Einheit, eben einem Paket, zu gruppieren. Pakete sind nützlich, um große Sammlungen von Klassen und Schnittstellen zu organisieren und mögliche Namenskonflikte zu verhindern. Weiterhin dienen sie zur Strukturierung der Sichtbarkeit von Klassen.

Paket

23.1 Vereinbarung von Paketen

Definition 23.1-1: Vereinbarung eines Pakets

Ein Paket (engl. package) wird wie folgt eingeführt:

```
package paketName;
```

Diese Anweisung muss am Anfang einer Übersetzungseinheit (z. B. einer einzelnen Quelldatei) vor die Klassendeklaration gesetzt werden, die zum Paket paketName gehören soll. [JLS: § 7]

package

Üblicherweise werden die Elemente eines Pakets in einem gleichnamigen Verzeichnis (engl. *directory*) abgelegt.

Pakete können Teile anderer Pakete sein. Wenn Pakete verschachtelt sind, erwartet der Java-Übersetzer, dass die Verzeichnisstruktur, in der die Klassen und Pakete abgelegt sind, mit der Pakethierarchie übereinstimmt.

Beispiel 23.1-1: Paket

Eine vollständige Anwendung „Blumenladenverwaltung“ könnte beispielsweise in folgende Pakete aufgeteilt sein:

- Ein Paket *kunde*, das die Klassen zur Verwaltung von Kunden einschließlich Rechnungen, Bestellungen usw. enthält.
- Ein Paket *waren*, das die Klassen zum Einkauf und Verkauf von Artikeln enthält.
- Ein Paket *laden*, das die übergeordnete Anwendungslogik für die verschiedenen Anwendungsfälle enthält.
- Ein Paket *gui*, das Komponenten der grafischen Benutzungsschnittstelle enthält.

Alle Klassen im Paket kunde beginnen mit der Anweisung:

```
package kunde;
```

um so dem Java-Übersetzer mitzuteilen, zu welchem Paket die Klasse gehört.

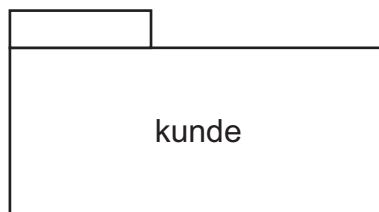


Abb. 23.1-1: Das Paket kunde

UML-Darstellung von
Paketen

Abb. 23.1-1 zeigt das Paket Kunde in UML-Darstellung.

☞ Nach der Konvention enthalten Paketnamen keine Großbuchstaben.

Standardpaket

Wenn für eine Klasse kein Paket angegeben ist, gehört sie mit allen Klassen des aktuellen Verzeichnisses zu einem sogenannten Standardpaket (engl. *default package*). Damit befindet sich grundsätzlich jede Klasse in einem Paket. [JLS: § 7.4.2]

23.2 Klassennamen und Import

Für die folgenden Betrachtungen gehen wir davon aus, dass alle Beispielklassen mit Hilfe von Zugriffsmodifikatoren über Paketgrenzen hinaus sichtbar und zugreifbar gemacht worden sind. Ohne den Einsatz von Zugriffsmodifikatoren können nur Klassen, die sich in demselben Paket befinden, aufeinander zugreifen. Zugriffsmodifikatoren und die Beeinflussung der Sichtbarkeit, auch über Paketgrenzen hinweg, sind Gegenstand von Kapitel 24.

Gültigkeitsbereich einer
Klasse

Der Gültigkeitsbereich eines einfachen Klassennamens wie `Rechnung` erstreckt sich auf das Paket, in dem sich die Klasse `Rechnung` befindet. Gehört die Klasse `Rechnung` zu einem Paket `kunde`, so können alle anderen Klassen aus dem Paket `kunde` die Klasse `Rechnung` mit ihrem einfachen Namen ansprechen. Eine Klasse, die nicht zum Paket `kunde` gehört, muss hingegen einen vollständigen Namen (engl. *qualified name*), zusammengesetzt aus dem Paketnamen und dem Klassennamen, verwenden [JLS: § 6.7]:

vollständiger Name

```
kunde.Rechnung
```

Gäbe es eine weitere Klasse `Rechnung` in einem anderen Paket, sagen wir `xy`, so gäbe es keine Namenskollision, da auf diese Klasse mit dem Namen

```
xy.Rechnung
```

verwiesen würde.

Eine explizite Benennung des Paketnamens für eine externe Klasse ist normalerweise nicht erforderlich, wenn wir eine Klasse mit einer Importanweisung wie folgt einbinden [JLS: § 7.5]:

```
import kunde.Rechnung;
```

Die Importanweisung muss nach der Paketanweisung und vor der Klassendefinition platziert werden. Durch eine Importanweisung wird der Name der importierten Klasse sichtbar gemacht. Die importierte Definition muss nun nicht mehr mit vollständigem Namen angesprochen werden, der Klassenname genügt.

Wenn mehr als eine Klasse aus einem Paket importiert werden muss, können wir Klassen nach Bedarf einbinden, indem wir folgende Importanweisung benutzen [JLS: § 7.5.2]:

```
import kunde.*;
```

Nun werden alle Namen des importierten Pakets (z. B. kunde) automatisch importiert.

Beispiel 23.2-1: Import von Paketen

Eine Klasse Bestellung im Paket kunde könnte folgende Importanweisungen enthalten:

```
package kunde;

import waren.Artikel;
import waren.ArtikelListe;

class Bestellung {
    // ...
}
```

oder alle Klassen des Pakets waren gemeinsam importieren:

```
package kunde;

import waren.*;

class Bestellung {
    // ...
}
```

└

Bemerkung 23.2-1: Unterpakete

Beachten Sie, dass sich die Importanweisung mit Platzhalter nicht automatisch auf Unterpakete erstreckt. Klassen in einem Unterpaket b eines Pakets a werden durch die Anweisung

```
import a.b.*;
```

importiert. Sollen sowohl Klassen aus dem Paket a , als auch Klassen aus dem Paket $a.b$ importiert werden, so sind zwei Importanweisungen erforderlich:

```
import a.*;  
import a.b.*;
```

└

Selbsttestaufgabe 23.2-1:

Stellen Sie fest, ob der folgende Programmtext gültig ist, unter der Annahme, dass sich die einzelnen Klassendefinitionen in den jeweils richtigen Dateien befinden.

```
package a;  
class A {}  
  
package a;  
class B {}  
  
package b.c;  
class C {}  
  
package d;  
import a.B;  
import b.*;  
class D {  
    A myA = new A();  
    C myC = new C();  
}
```

◇

24 Geheimnisprinzip und Zugriffskontrolle

Nachdem wir nun Pakete und Vererbung als weitere Bestandteile der objektorientierten Programmierung in Java kennen gelernt haben, wird es Zeit, sich mit dem Geheimnisprinzip und den Sprachmitteln, die dessen Umsetzung in Java ermöglichen, auseinander zu setzen.

24.1 Geheimnisprinzip

In der Programmierung versucht man das Verhalten von der konkreten Implementierung zu trennen. Für die Verwendung eines Programms bzw. einer Klasse muss lediglich bekannt sein, was für eine Funktionalität geboten wird, aber nicht wie genau sie umgesetzt ist. Dieses Konzept nennen wir auch Geheimnisprinzip, da die genaue Implementierung für Außenstehende ein Geheimnis ist. Dieses Prinzip bringt eine Reihe von Vorteilen mit sich. Da die Interna der Klasse nach außen hin nicht bekannt sind, verursachen Änderungen an der Implementierung keine Probleme, solange die vereinbarte Funktionalität beibehalten wird.

Geheimnisprinzip

Bisher haben wir den direkten Zugriff auf Attribute vermieden, um so unsere Klassen robuster gegen Änderungen zu gestalten. Ein Benutzer der Klasse ist durch die Verwendung von Getter- und Setter-Methoden unabhängig von der genauen internen Implementierung der Attribute. Wir haben bisher versucht uns an dieses Prinzip zu halten, konnten aber nicht verhindern, dass von außen auf Attribute zugegriffen werden konnte.

Auch bei Methoden kann es sinnvoll sein, den Zugriff von außen einzuschränken, wenn es sich zum Beispiel lediglich um interne Hilfsmethoden handelt. Ebenso kann es sein, dass man den Zugriff auf eine Klasse von außerhalb eines Pakets verhindern will.

Um den Zugriff auf Klassen und ihre Elemente zu beschränken bietet Java sogenannte Zugriffsmodifikatoren. Wir werden sie und ihre Bedeutung in den folgenden Abschnitten kennen lernen.

24.2 Zugriffskontrolle bei Paketen und Klassen

In größeren Projekten werden Klassen in Paketen organisiert. Die Pakete selbst sind immer sichtbar [JLS: § 6.6.1]. Bei Klassen gibt es zwei Möglichkeiten. Sie können entweder in allen Paketen zur Verfügung gestellt werden, oder sie sind nur im eigenen Paket sichtbar. Die Zugriffskontrolle bei Klassen erfolgt über Zugriffsmodifikatoren, die dem Schlüsselwort `class` vorangestellt werden.

Zugriffsmodifikatoren

Definition 24.2-1: Zugriffskontrolle bei eigenständigen Klassen

Zugriffskontrolle bei
eigenständigen Klassen

Eine Klasse ist öffentlich (engl. public) zugänglich, falls in der Klassendefinition dem Schlüsselwort `class` das Schlüsselwort `public` vorangestellt ist:

```
public class className {
    ...
}
```

`public` *Eine öffentliche Klasse ist in allen Paketen sichtbar.*

Standardzugriff *Falls `public` fehlt, so ist lediglich der Standardzugriff (engl. default access) von innerhalb des Pakets, zu dem die Klasse gehört, möglich.*

Andere Zugriffsmodifikatoren sind bei eigenständigen Klassen nicht zulässig. [JLS: § 6.6.1]

┘

Beispiel 24.2-1: Zugriff auf Klassen

Gegeben seien die Pakete `a`, `a.c` und `b` mit den folgenden Klassendeklarationen.

```
package a;

class K {
    // Hier sind die Klassen
    // L, M und N sichtbar
}

public class L {
    // Hier sind die Klassen
    // K, M und N sichtbar
}

package a.c;

public class M {
    // Hier sind die Klassen
    // L und N sichtbar
}

package b;

public class N {
    // Hier sind die Klassen
    // L, M und P sichtbar
}

class P {
    // Hier sind die Klassen
    // L, M und N sichtbar
}
```

Die Klasse `K` ist nur im Paket `a` sichtbar. Das Paket `a.c` ist kein Bestandteil von `a`, somit ist `K` dort nicht sichtbar. Die Klasse `P` ist nur im Paket `b` sichtbar.

┘

Eine als `public` deklarierte Klasse `X` muss immer in einer Datei `X.java` gespeichert sein.

24.3 Zugriffskontrolle bei Klassenelementen

Zusätzlich zur Zugriffskontrolle auf Klassen kann auch der Zugriff auf die Elemente einer Klasse, zum Beispiel Attribute, Methoden und Konstruktoren, eingeschränkt werden. Auch dieser Zugriff wird durch Zugriffsmodifikatoren festgelegt.


Definition 24.3-1: Zugriffskontrolle bei Attributen

Bei der Attributdeklaration kann dem Typen noch ein Zugriffsmodifikator vorangestellt werden. [JLS: § 8.3.1]

Zugriffskontrolle bei Attributen

```
Zugriffsmodifikator Typ Name;
```

└

 Treten bei einer Attributdeklaration sowohl ein Zugriffsmodifikator als auch das Schlüsselwort `final` oder `static` auf, so wird zuerst der Zugriffsmodifikator und anschließend `static` und dann `final` genannt.


Definition 24.3-2: Zugriffskontrolle bei Methoden

Bei der Methodendeklaration kann dem Ergebnistyp noch ein Zugriffsmodifikator vorangestellt werden. [JLS: § 8.4.3]

Zugriffskontrolle bei Methoden

```
Zugriffsmodifikator Ergebnistyp Name(Parameterliste) {  
}
```

└

 Treten bei einer Methodendeklaration sowohl ein Zugriffsmodifikator als auch das Schlüsselwort `final` oder `static` auf, so wird zuerst der Zugriffsmodifikator und anschließend `static` und dann `final` genannt.

Definition 24.3-3: Zugriffskontrolle bei Konstruktoren

Bei der Konstruktordeklaration kann dem Namen noch ein Zugriffsmodifikator vorangestellt werden. [JLS: § 8.8.3]

Zugriffskontrolle bei Konstruktoren

```
Zugriffsmodifikator Name(Parameterliste) {  
}
```

└

Zusätzlich zum Zugriffsmodifikator `public`, den wir schon bei Klassen kennen gelernt haben, und dem Standardzugriff ohne expliziten Zugriffsmodifikator gibt es bei Klassenelementen die beiden Modifikatoren `protected` und `private`.

Definition 24.3-4: Öffentliche (`public`) Klassenelemente

Ein Klasselement, dass als `public` deklariert ist, ist überall dort sichtbar, wo auch die Klasse sichtbar ist.

öffentlich
`public`

└

Bemerkung 24.3-1: Öffentliche Klassenelemente

In der Regel werden Methoden und Konstruktoren sowie Konstanten, die für jeden Benutzer der Klasse zur Verfügung stehen sollen, als `public` deklariert.

Bemerkung 24.3-2: `main()`-Methode

`main()`-Methode

Mit unseren jetzigen Kenntnissen ist es uns auch möglich, alle Bestandteile der `main()`-Methode zu verstehen.

Die `main()`-Methode ist eine öffentliche (`public`) Klassenmethode (`static`), die kein Ergebnis zurück gibt (`void`) und als einziges Argument ein Zeichenkettenfeld (`String[] args`) akzeptiert.

Soll im Programm auf die übergebenen Argumente zugegriffen werden, so kann dies über das Zeichenkettenfeld geschehen. Werden zum Beispiel Zahlen übergeben, so müssen diese mit geeigneten Methoden der API in Werte primitiver Datentypen konvertiert werden.

Definition 24.3-5: Standardzugriff bei Klassenelementen

Standardzugriff bei
Klassenelementen

Ein Klassenelement, dessen Deklaration keinen Zugriffsmodifikator enthält, ist innerhalb des Paketes sichtbar, in dem die zugehörige Klasse enthalten ist. [JLS: § 6.6.1]

Definition 24.3-6: Private (`private`) Klassenelemente

`privat`
`private`

Ein Klassenelement, dass als `private` deklariert ist, ist nur innerhalb der Klasse sichtbar.

Bemerkung 24.3-3: Private Klassenelemente

Attribute werden in der Regel als `private` deklariert, damit sie so von außerhalb der Klasse nicht direkt verändert werden können. Auch Hilfsmethoden werden in der Regel als `private` deklariert.

Wir können nun unsere Klassen `Artikel` und `Pflanze` um die entsprechenden Zugriffsmodifikatoren ergänzen. Dabei wollen wir erreichen, dass die Klassen sowie die Getter-Methoden und die Methode `gebeInformationenAus()` überall sichtbar sind. Attribute hingegen sollen nur in der Klasse sichtbar sein und Konstruktoren und Setter-Methoden nur innerhalb des Pakets waren.

```
// Inhalt der Datei Artikel.java
package waren;

public class Artikel {
    private double preis;
    private String name;
    private int artikelnr;

    Artikel(final double preis, final int artikelnr) {
        this.legePreisFest(preis);
        this.legeArtikelnummerFest(artikelnr);
    }

    Artikel(final double preis, final int artikelnr,
            final String name) {
        this(preis, artikelnr);
        this.legeNameFest(name);
    }

    public double lieferePreis() {
        return this.preis;
    }

    void legePreisFest(final double neuerPreis) {
        this.preis = neuerPreis;
    }

    public String liefereName() {
        return this.name;
    }

    void legeNameFest(final String neuerName) {
        this.name = neuerName;
    }

    public int liefereArtikelnummer() {
        return this.artikelnr;
    }

    void legeArtikelnummerFest(final int neueNr) {
        this.artikelnr = neueNr;
    }

    public void gebeInformationenAus() {
        System.out.print("Artikel ");
        System.out.println(this.liefereName());
        System.out.print("Nr: ");
        System.out.println(this.liefereArtikelnummer());
        System.out.print("Preis: ");
        System.out.println(this.lieferePreis());
    }
}
```

```
// Inhalt der Datei Pflanze.java
package waren;

public class Pflanze extends Artikel {
    private double lagertemperatur;

    Pflanze(final double preis, final int artikelnr,
            final double lagertemp) {
        super(preis, artikelnr);
        this.legeLagertemperaturFest(lagertemp);
    }

    Pflanze(final double preis, final int artikelnr,
            final String name, final double lagertemp) {
        super(preis, artikelnr, name);
        this.legeLagertemperaturFest(lagertemp);
    }

    public double liefereLagertemperatur() {
        return this.lagertemperatur;
    }

    void legeLagertemperaturFest(final double temp) {
        this.lagertemperatur = temp;
    }

    public void gebeInformationenAus() {
        super.gebeInformationenAus();
        System.out.print("Lagertemperatur: ");
        System.out.println(this.liefereLagertemperatur());
    }
}
```

Definition 24.3-7: Geschützte (protected) Klassenelemente

geschützt
protected

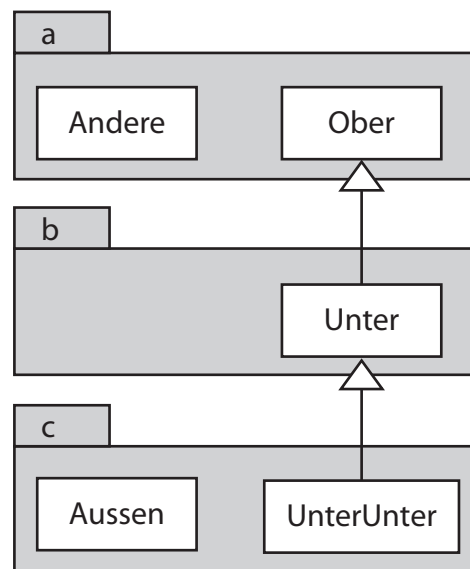
Ein Klassenelement, dass als `protected` deklariert ist, ist innerhalb des Paketes der Klasse sichtbar. Eine Unterklasse U hat auch außerhalb des Pakets der Oberklasse O Zugriff auf dieses Element. Dazu muss jedoch O als öffentlich deklariert sein, und der Zugriff muss an einem Element vom Typ U oder einem Subtyp von U ausgeführt werden. [JLS: § 6.6.1, § 6.6.2]

└

Bemerkung 24.3-4: Geschützte Klassenelemente

Methoden und Attribute, die nicht jedem Benutzer zur Verfügung stehen sollen, jedoch von Unterklassen benötigt werden, werden in der Regel als `protected` deklariert.

└

Beispiel 24.3-1: Zugriff auf geschützte Klasselemente**Abb. 24.3-1:** Klassenhierarchie

```

package a;

public class Ober {
    protected int x;
}

public class Andere {
    public void test() {
        Ober o = new Ober();
        // Zugriff innerhalb des Paketes möglich
        o.x = 10;
    }
}

package b;

public class Unter extends a.Ober {
    public void test() {
        // Zugriff auf geerbtes Attribut möglich
        x = 9;
        a.Ober o = new a.Ober();
        // Zugriff nicht möglich da anderes Paket und Typ oberhalb
        // in der Vererbungshierarchie
        // o.x = 10;
        a.Ober ou = new Unter();
        // Zugriff nicht möglich da anderes Paket und da Typ der
        // Variable und nicht des Objekts entscheidend
        // ou.x = 10;
        Unter u = new Unter();
        // Zugriff bei Variablen der gleichen Klasse möglich
        u.x = 99;
    }
}
  
```

```

        c.UnterUnter uu = new c.UnterUnter();
        // Zugriff bei Variablen einer Unterklasse möglich
        uu.x = 7;
    }
}

package c;

public class UnterUnter extends b.Unter {
    public void test() {
        // Zugriff auf geerbtes Attribut möglich
        x = 9;
        b.Unter u = new b.Unter();
        // Zugriff nicht möglich da Typ oberhalb in der
        // Vererbungshierarchie
        // u.x = 99;
        UnterUnter uu = new UnterUnter();
        // Zugriff bei Variablen der gleichen Klasse möglich
        uu.x = 7;
    }
}

public class Aussen {
    public void test() {
        a.Ober o = new a.Ober();
        // Zugriff außerhalb des Paketes und der
        // Vererbungshierarchie nicht möglich
        // o.x = 10;
        UnterUnter uu = new UnterUnter();
        // Zugriff außerhalb des Paketes und der
        // Vererbungshierarchie nicht möglich
        // uu.x = 7;
    }
}

```

Wäre die Klasse Ober nicht öffentlich, sondern ohne Zugriffsmodifikator deklariert, so wäre ein Zugriff auf Ober und x nur von innerhalb des Paketes a möglich, auch wenn x als geschützt deklariert wäre. Die Klasse b.Unter könnte in diesem Fall auch keine Unterklasse von a.Ober sein, da a.Ober nicht sichtbar wäre. ┘

Zugriffsmodifikatoren in
der UML

Die Zugriffsmodifikatoren für Klassenelemente können auch in der UML dargestellt werden. Die UML bietet wie Java vier verschiedene Modifikatoren (VisibilityKind): public, private, protected und package. Dabei entspricht package dem Standardzugriff von Java. Die Bedeutung von protected unterscheidet sich jedoch, da in der UML dieser Modifikator andeutet, dass nur Unterklassen Zugriff auf dieses Element haben, Klassen im gleichen Paket jedoch nicht. Die Modifikatoren werden in der UML dem Methodennamen vorangestellt und durch die folgenden Zeichen dargestellt:

public	+
private	-
protected	#
package	~

Die beiden Klassen `Artikel` und `Pflanze` sind in Abb. 24.3-2 in Form eines UML-Klassendiagramm dargestellt.

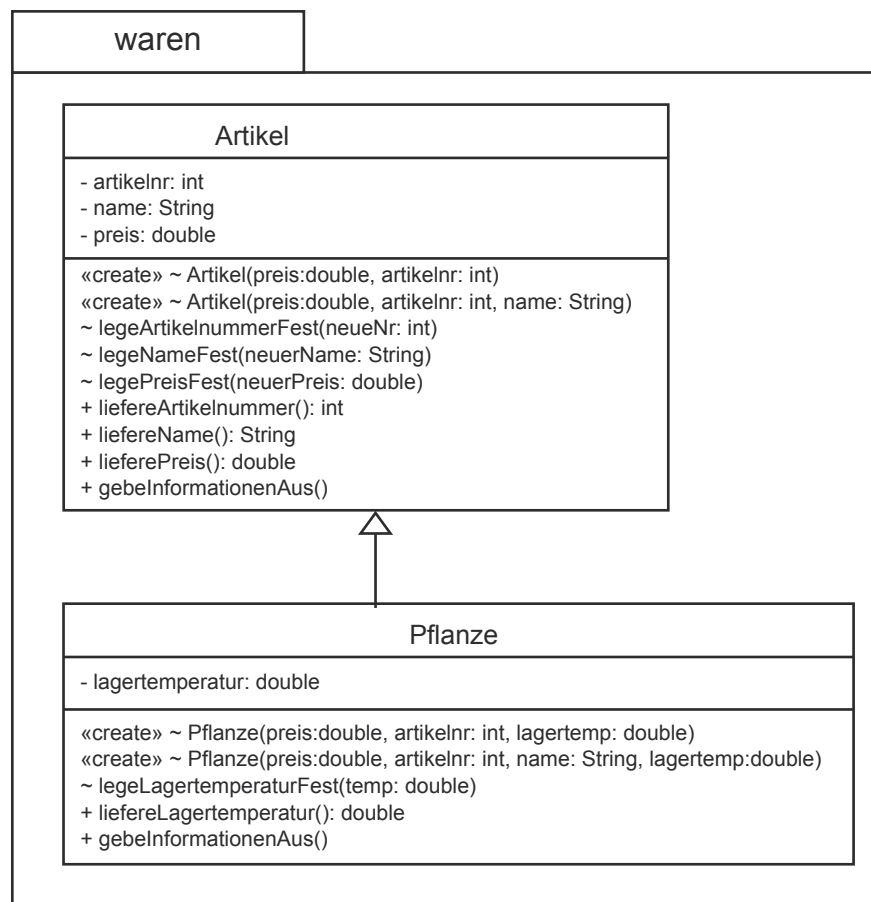


Abb. 24.3-2: Zugriffsmodifikatoren in der UML

Beim Überschreiben von Methoden darf der Zugriff in der Unterklasse höchstens erweitert, aber niemals eingeschränkt werden. d. h. der Zugriffsmodifikator der Methode in der Unterklasse muss größer oder gleich dem der Oberklasse sein [JLS: § 8.4.8.3]. Dabei gilt die folgende Ordnung:

public > protected > Standardzugriff > private.

Da wir die Bedeutung der verschiedenen Zugriffsmodifikatoren nun kennen, können wir verstehen, welche Elemente vererbt werden.

Für die Vererbung gelten für Klassen folgende Regeln:

- Eine Unterklasse erbt alle öffentlichen Attribute und Methoden der Oberklasse,

Zugriffskontrolle bei überschriebenen Methoden

Vererbungsregeln

- alle geschützten Attribute und Methoden der Oberklasse sowie
- alle Attribute und Methoden ohne Zugriffsmodifikator der Oberklasse, wenn sich die Oberklasse im gleichen Paket befindet.

Konstruktoren werden nicht vererbt.

Eine finale Klasse kann nicht erweitert, also auch nicht beerbt werden.

Da eine Oberklasse selbst Attribute und Methoden von ihrer Oberklasse geerbt haben kann, erhalten wir Vererbungsketten, die von der obersten Klasse bis zu den Blättern einer Klassenhierarchie reichen.

Lokale Variablen, d. h. Variablen, die innerhalb von Methoden deklariert sind, sind nicht von außerhalb der jeweiligen Methode und somit auch niemals von außerhalb der Klasse zugänglich.

Selbsttestaufgabe 24.3-1:

Analysieren Sie die folgenden Klassen und entscheiden Sie, welche Zugriffe (Z1 bis Z10) zulässig sind, solange f als public deklariert ist, und welche zulässig sind, wenn der Zugriffsmodifikator von f zu private oder protected wechselt oder ganz entfällt.

```
package x;

public class A {
    public int f;

    public void test() {
        y.C c1 = new y.C();
        c1.f = 6;           // Z1
        f = 8;             // Z2
    }
}

public class B {
    public void test() {
        A a1 = new A();
        a1.f = 10;         // Z3
        f = 3;             // Z4
    }
}

package y;

public class C extends x.A {
    public void test() {
        x.A a2 = new x.A();
        a2.f = 1;         // Z5
        C c2 = new C();
    }
}
```



```

        c2.f = 9;           // Z6
        f = 7;             // Z7
    }
}

public class D extends x.B {
    public void test() {
        x.A a3 = new x.A();
        a3.f = 1;          // Z8
        C c3 = new C();
        c3.f = 9;          // Z9
        f = 7;             //Z10
    }
}

```



Selbsttestaufgabe 24.3-2:

Gegeben seien folgende Klassendeklarationen:

```

class A {
    public void m(int x, String y) {
        System.out.println("A: " + x + " " + y);
    }
}

class B extends A {
    public void m(int x) {
        System.out.println("B: " + x);
    }
}

class C extends A {
    public void m(String x, String y) {
        System.out.println("C: " + x + " " + y);
    }

    private void n(int x) {
        System.out.println("C: " + x);
    }
}

class D extends C {
    public void m(String x, String y) {
        System.out.println("D: " + x + " " + y);
    }
}

class E extends C {
    public void m(String x) {
        System.out.println("E: " + x);
    }
}

```

```

        public void n(boolean x) {
            System.out.println("E: " + x);
        }
    }

    class F extends E {
        public void m(int x) {
            System.out.println("F: " + x);
        }
    }

    class G extends E {
        public void m(String x) {
            System.out.println("G: " + x);
        }
    }

```

1. Stellen Sie die Klassenhierarchie in Form eines UML-Klassendiagramms dar.
2. Weiterhin wollen wir annehmen, der folgende Code sei in einer Klasse, für die alle oben aufgeführten Klassen sichtbar sind, enthalten:

```

    C c = new C();           // [1]
    E e = new E();           // [2]
    F f = new F();           // [3]
    G g = new G();           // [4]
    g.m("hello");            // [5]
    e.m("hello");            // [6]
    g.n(true);               // [7]
    g.m("hello", "guys");    // [8]
    g.n(3);                  // [9]
    g.m(2, "times");         // [10]
    f.m(100);               // [11]
    g.m(2);                  // [12]
    f.m(4, "you");           // [13]
    e.m(8);                 // [14]
    c.n(6);                 // [15]

```

- An welchen Stellen gibt es aus welchen Gründen Fehlermeldungen bei der Übersetzung?
- Welcher Text wird bei der Ausführung dieser Anweisungen ausgegeben, wenn die fehlerhaften Zeilen entfernt werden?



25 Abstrakte Einheiten

Klassen nahe der Wurzel einer Vererbungshierarchie werden oft so allgemein gehalten, dass es keinen sinnvollen Weg gibt, alle Methoden und Attribute dieser Klasse bereits auf dieser Ebene der Abstraktion zu implementieren.

Java bietet zwei Konzepte an, um mit solchen Fällen umzugehen: Abstrakte Klassen und Schnittstellen. Beide Konzepte dienen vor allem dem Zweck, Eigenschaften und Verhaltensweisen zu vereinbaren, die in allen Unterklassen einer abstrakten Einheit verfügbar gemacht werden sollen, ohne sie bereits zu implementieren. Beide Konzepte weisen aber auch Unterschiede auf, die wir im Folgenden erörtern wollen.

25.1 Abstrakte Klassen und Methoden

Eine abstrakte Klasse sorgt dafür, dass ihre Unterklassen bestimmte Eigenschaften aufweisen und einheitliche Dienste nach außen zur Verfügung stellen.

Definition 25.1-1: Abstrakte Klasse

Eine abstrakte Klasse wird durch das vorangestellte Schlüsselwort `abstract` vereinbart. [JLS: § 8.1.1.1]

abstrakte Klasse
abstract bei Klassen

Von abstrakten Klassen können keine Exemplare erzeugt werden.

Eine abstrakte Klasse kann abstrakte Methoden besitzen.

┘

Eine Methode, für die keine Implementierung angegeben wird, heißt abstrakte Methode.

Definition 25.1-2: Abstrakte Methode

Eine abstrakte Methode weist nur eine Signatur und einen Ergebnistyp oder `void` auf und wird durch das vorangestellte Schlüsselwort `abstract` vereinbart. [JLS: § 8.4.3.1]

abstrakte Methode
abstract bei
Methoden

Die Vereinbarung einer abstrakten Methode wird mit einem Semikolon abgeschlossen:

```
Zugriffsmodifikator abstract Ergebnistyp Name(Parameterliste);
```

┘

Wenn in einer Klasse eine oder mehrere Methoden als `abstract` vereinbart sind, muss die Klasse auch als `abstract` vereinbart sein.

Eine Klasse kann jedoch auch als `abstract` erklärt werden, wenn keine ihrer Methoden als `abstract` deklariert ist.

Für abstrakte Klassen gelten folgende Vererbungsregeln:

- Jede konkrete Klasse, die von einer abstrakten Klasse erbt, muss die von der abstrakten Klasse geerbten abstrakten Methoden überschreiben und implementieren.
- Wenn eine oder mehrere abstrakte Methoden der abstrakten Oberklasse in der Unterklasse nicht implementiert werden, dann bleiben diese Methoden abstrakt, und die Unterklasse muss auch abstrakt sein.

In der UML wird eine Klassen als abstrakt gekennzeichnet, indem ihr Name kursiv geschrieben wird. Alternativ kann zur Kennzeichnung hinter oder unter dem Klassennamen `{abstract}` ergänzt werden.

Abstrakte Klassen sind nützlich, wenn wir für manche Methoden Implementierungen angeben wollen und für andere nicht. Die Implementierung einer abstrakten Methode in verschiedenen Unterklassen muss mit der Vereinbarung der abstrakten Methode übereinstimmen.

Die Vereinbarung abstrakter Methoden empfiehlt sich, wenn für eine Methode keine sinnvolle Standardimplementierung in einer Oberklasse angegeben werden kann.

Beispiel 25.1-1: Pflanzgefäße und Blumenerde

Als Service für Käufer von Blumentöpfen und anderen Pflanzgefäßen soll das Volumen der insgesamt benötigten Blumenerde berechnet werden. Wir verwenden eine kleine Klassenhierarchie mit verschiedenen Arten von Pflanzgefäßen als Subklassen einer abstrakten Oberklasse `Pflanzgefaess`. Abb. 25.1-1 zeigt die Subklassen `Topf` für konische Blumentöpfe und `Kiste` für quaderförmige Blumenkisten.

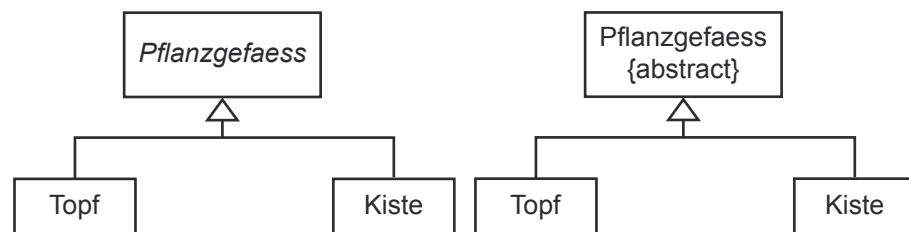


Abb. 25.1-1: Die abstrakte Klasse `Pflanzgefaess` (in zwei möglichen Darstellungen) mit den Subklassen `Topf` und `Kiste`

Die Gefäße, die eine Kundin bestellt hat, könnten beispielsweise in einem Feld

```
Pflanzgefaess[] bestellteGefaessee;
```

zusammengefasst sein. Die Typen der Elemente von `bestellteGefaessee` sind Subtypen von `Pflanzgefaess`. Um zu berechnen, wie viel Blumenerde insgesamt benötigt wird, sollen die Volumina aller Elemente des Feldes `bestellteGefaessee` wie folgt addiert werden:

```
double volumen = 0;
for (Pflanzgefaess gefaess : bestellteGefaessee) {
    volumen += gefaess.berechneVolumen();
}
```

Dazu muss die Klasse Pflanzgefaess eine Methode berechneVolumen() anbieten. Da das Volumen für konische Töpfe und für quaderförmige Kisten aber ganz unterschiedlich berechnet wird, kann eine sinnvolle Implementierung nur in den jeweiligen Unterklassen erfolgen. Auch die zur Berechnung benötigten Maße (Attribute) unterscheiden sich je nach Geometrie der Gefäße. Die abstrakte Klasse Pflanzgefaess implementiert deshalb die Methode berechneVolumen() nicht, sondern legt durch die Definition einer abstrakten Methode berechneVolumen() nur fest, dass alle konkreten Subklassen eine solche Methode implementieren müssen. Daneben definiert sie bereits einige allgemeine Attribute und Methoden.

```
public abstract class Pflanzgefaess extends Artikel {

    private int farbcode;
    private double hoehe;
    // ... (weitere Attribute)
    protected final double PI = 3.14159265;

    protected void legeFarbeFest(final int farbe) {
        this.farbcode = farbe;
    }

    // ... (weitere Methoden)

    public abstract double berechneVolumen();
}
```

Die konkreten Subklassen Topf und Kiste ergänzen die Klasse Pflanzgefaess um zusätzliche Attribute und Methoden und implementieren die Methode berechneVolumen():

```
public class Topf extends Pflanzgefaess {

    private double durchmesserOben;
    private double durchmesserUnten;
    // ... (weitere Attribute)

    public Topf (final double hoehe, final double durchmesserOben,
                final double durchmesserUnten) {
        this.legeHoeheFest(hoehe);
        this.legeDurchmesserObenFest(durchmesserOben);
        this.legeDurchmesserUntenFest(durchmesserUnten);
    }

    public double berechneVolumen() {
        double radOben = this.liefereDurchmesserOben() / 2;
        double radUnten = this.liefereDurchmesserUnten() / 2;
        double volumen = (radOben * radOben
                        + radOben * radUnten
                        + radUnten * radUnten)
                        * this.liefereHoehe() * PI / 3;
    }
}
```

```

        return volumen;
    }

    // ... (weitere Methoden)
}

public class Kiste extends Pflanzgefaess {

    private double breite;
    private double laenge;
    // ... (weitere Attribute)

    public Kiste(final double hoehe, final double breite,
                 final double laenge) {
        this.legeHoeheFest(hoehe);
        this.legeBreiteFest(breite);
        this.legeLaengeFest(laenge);
    }

    public double berechneVolumen() {
        return this.liefereBreite()
            * this.liefereLaenge()
            * this.liefereHoehe();
    }

    // ... (weitere Methoden)
}

```

Der Versuch, ein Exemplar einer abstrakten Klasse zu erzeugen, wird einen Übersetzungsfehler hervorrufen. Wenn wir hingegen schreiben

```
Pflanzgefaess meinGefaess = new Topf(.3, .35, .25);
```

erzeugen wir ein Exemplar der Klasse `Topf`, das einer Verweisvariablen vom Typ `Pflanzgefaess` zugeordnet wird; wir erzeugen jedoch kein `Pflanzgefaess`-Objekt.

Bisweilen sind abstrakte Klassen nützlich, um zu gewährleisten, dass nur Objekte von Unterklassen erzeugt werden.

Beispiel 25.1-2: Abstrakte Klasse `Artikel`

Da jeder Artikel in einem Blumenladen einer möglichst spezifischen Warengruppe angehören sollte, ist es durchaus sinnvoll, die Erzeugung von Objekten allgemeiner Oberklassen zu verhindern. So könnten etwa die Klasse `Artikel` oder auch andere Oberklassen wie `Pflanze` als `abstract` deklariert werden.

Ein Konstruktor kann nicht als `abstract` deklariert werden. Da Konstruktoren nicht überschrieben werden können, gibt es keine Möglichkeit, einen Konstruktor erst in einer abgeleiteten Klasse zu implementieren. Eine abstrakte Klasse darf aber durchaus Konstruktoren implementieren, die dann von Unterklassen mit Hilfe von `super()` verwendet werden können.

Beispiel 25.1-3: Abstrakte Klasse GreetingCard

Das folgende Programm zeigt, dass auch in der abstrakten Klasse schon die abstrakten Methoden von anderen Methoden aus aufgerufen werden können. Auch wird deutlich, dass Konstruktoren in abstrakten Klassen hilfreich sein können. Überlegen Sie, welche Ausgabe die main()-Methode der Klasse GreetingCard erzeugt.

```
public abstract class GreetingCard {

    private String sender;

    public GreetingCard(final String sender) {
        this.sender = sender;
    }

    public void print() {
        printGreeting();
        System.out.println("Best regards");
        System.out.println(sender);
    }

    protected abstract void printGreeting();

    public static void main(String[] args) {
        GreetingCard card = new ChristmasCard("Franz");
        card.print();
        card = new BirthdayCard("Hans", "Franz");
        card.print();
    }
}

public class BirthdayCard extends GreetingCard {

    private String name;

    public BirthdayCard(final String sender, final String name) {
        super(sender);
        this.name = name;
    }

    protected void printGreeting() {
        System.out.print("Happy Birthday, ");
        System.out.print(name);
        System.out.println("!");
    }
}
```

```

public class ChristmasCard extends GreetingCard {

    public ChristmasCard(final String sender) {
        super(sender);
    }

    protected void printGreeting() {
        System.out.print("Merry Christmas ");
        System.out.println("and a Happy New Year!");
    }
}

```

└

Als `private` deklarierte Methoden können nicht als `abstract` vereinbart werden. Der Grund für diese Einschränkung liegt auf der Hand: sie könnten in einer abgeleiteten Klasse nicht überschrieben werden, weil `private` Methoden und Attribute außerhalb der definierenden Klasse nicht sichtbar sind.

Statische Methoden können ebenfalls nicht als `abstract` vereinbart werden.

Java lässt es auch nicht zu, eine Klasse als `final` und `abstract` zu deklarieren. Wenn eine Klasse als `final` vereinbart wird, drücken wir damit aus, dass von dieser Klasse keine Unterklassen abgeleitet werden können. Damit besteht auch keine Möglichkeit, abstrakte Methoden in dieser Klasse zu vereinbaren. Analog können auch Methoden nicht als `final` und `abstract` deklariert werden.

☞ Treten bei einer Klasse oder bei einer Methode die Modifikatoren `public` und `abstract` gemeinsam auf, so wird zuerst `public` genannt.

Selbsttestaufgabe 25.1-1:

Enthält der nachstehende Programmausschnitt Fehler? Begründen Sie ihre Antwort!

```

class A {
    abstract int twoTimes(int i);
    abstract int fourTimes(int i) {
        return twoTimes(twoTimes(i));
    }
}

```

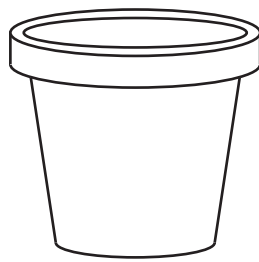
◇

Selbsttestaufgabe 25.1-2:

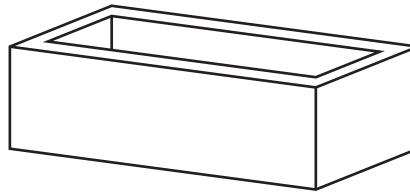
Schreiben Sie analog zu den Klassen `Topf` und `Kiste` aus Beispiel 25.1-1 zwei weitere konkrete Subklassen von `Pflanzgefaess` (vgl. Abb. 25.1-2):

- Zylinder für zylindrische Töpfe
- Wanne für liegende Halbzylinder

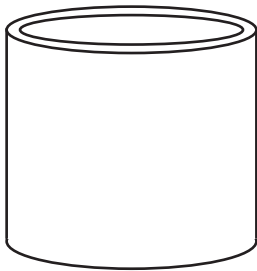
◇



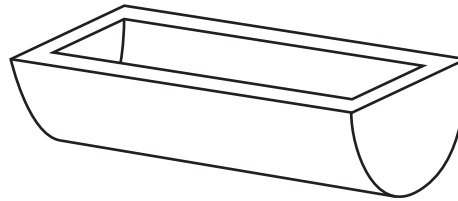
Topf



Kiste



Zylinder



Wanne

Abb. 25.1-2: Verschiedene Formen von Pflanzgefäßen**Selbsttestaufgabe 25.1-3:**

Definieren Sie die Klassen *Konto*, *Sparkonto* und *Girokonto*. *Sparkonto* und *Girokonto* sind Subklassen von *Konto*. Auf *Spar-* wie *Girokonten* kann beliebig viel Geld eingezahlt werden. Von *Sparkonten* kann maximal das vorhandene Guthaben abgeboben werden, *Girokonten* können im Rahmen eines (variablen) Dispositionskredits überzogen werden. Zinsen werden vernachlässigt.



25.2 Schnittstellen

Das Konzept der Schnittstellen (engl. *interface*) in Java ähnelt dem abstrakter Klassen in der Möglichkeit, Klassen zu zwingen, bestimmte Methoden zu implementieren. Es gibt jedoch den entscheidenden Unterschied, dass eine Schnittstelle keine Exemplarattribute kennt und keine Methoden implementiert.

Definition 25.2-1: Schnittstellenvereinbarung

Eine Schnittstelle wird wie folgt vereinbart [JLS: § 9.1]:

```
interface Schnittstellename {
    Attributdeklarationen
    Methodendeklarationen
}
```

Schnittstellen-
vereinbarung

Attributvereinbarungen können nur öffentliche finale Klassenattribute enthalten.

interface

Die Schlüsselwörter `public` `static` `final` sind zur Deklaration nicht erforderlich. [JLS: § 9.3]

Alle Methodendeklarationen sind öffentlich, nicht-statisch und abstrakt. Die Schlüsselwörter `public` `abstract` sind zur Deklaration nicht erforderlich. [JLS: § 9.4]

Der Rumpf der Schnittstelle darf leer sein.

`extends` bei
Schnittstellen

Eine Schnittstelle kann mit Hilfe des Schlüsselworts `extends` mehrere andere Schnittstellen erweitern. ┘

Die Zugriffskontrolle für Schnittstellen erfolgt nach denselben Regeln wie die Zugriffskontrolle bei Klassen (vgl. Abschnitt 24.2).

Für Schnittstellen gelten folgende Vererbungsregeln:

- Schnittstellen können von anderen Schnittstellen erweitert (engl. *extend*) werden. Sie können jedoch nicht durch Klassen erweitert werden.
- Klassen können Schnittstellen lediglich implementieren.
- Eine Klasse kann mehrere Schnittstellen implementieren.
- Schnittstellenmethoden müssen in einer Klassenimplementierung als `public` vereinbart werden, da alle Schnittstellenmethoden implizit `public` sind.
- Wenn Klasse A Schnittstelle B implementiert, ist A ein Subtyp von B. Somit kann ein Objekt vom Typ A überall dort verwendet werden, wo ein Objekt vom Typ B erwartet wird.
- Eine Klasse kann zugleich eine (oder mehrere) Schnittstelle(n) implementieren und eine Klasse erweitern.

Definition 25.2-2: Implementierung von Schnittstellen durch Klassen

Implementierung von
Schnittstellen
`implements`

Java benutzt das Schlüsselwort `implements`, um anzuzeigen, dass eine Klasse eine oder mehrere Schnittstellen implementiert [JLS: § 8.1.5]:

```
class Klassenname implements I [, I2, ..., In ] {
    ...
}
```

Eine Klasse muss jede Methode der Schnittstellen `I`, `I2`, ..., `In` mit den vereinbarten Ergebnistypen und Parametern implementieren oder, falls sie nicht alle Schnittstellenmethoden implementiert, als `abstract` gekennzeichnet sein. ┘

Schlüsselwort
«interface»

In UML-Klassendiagrammen werden Schnittstellen durch das Schlüsselwort «interface» gekennzeichnet. Eine `implements`-Beziehung wird durch einen Pfeil mit gestrichelter Linie und unausgefüllter Pfeilspitze ausgedrückt.

Einfachvererbung
Mehrfachvererbung

Bei Schnittstellen wird in Java das Prinzip der Einfachvererbung durchbrochen und eine Art von Mehrfachvererbung (engl. *multiple inheritance*) zugelassen. Mehrfach-

vererbung bedeutet, dass ein Typ mehrere Obertypen hat.

Bekanntlich ist jede Klasse ein Subtyp in einer Klassenhierarchie, entweder ein direkter Subtyp der Klasse `Object`, oder Subtyp einer spezielleren Klasse wie die Spezialisierungen der Klasse `Artikel`, die wir schon mehrfach als Beispiele verwendet haben. Eine Klasse, die eine Schnittstelle implementiert, ist zusätzlich auch ein Subtyp dieser Schnittstelle, was wir mit dem `instanceof`-Operator leicht überprüfen können:

Subtypenbildung bei Schnittstellen

```
public class A {}

public interface B {}

public interface BB {}

public interface BBB extends B, BB {}

public class C extends A implements BBB {
    public static void main(String[] args) {
        C myC = new C();
        System.out.println(myC instanceof A
                           && myC instanceof B
                           && myC instanceof BB
                           && myC instanceof BBB
                           && myC instanceof Object); // prints true
    }
}
```

Die Subtypbeziehung zwischen zwei durch Schnittstellen festgelegten Typen ist äquivalent zu der Subtypbeziehung zwischen Klassen definiert. Außerdem ist die Subtypbeziehung transitiv, d. h. wenn X ein Subtyp von Y ist und Y ein Subtyp von Z ist, dann ist X auch ein Subtyp von Z.

Mehrfachvererbung durch Schnittstellen lässt sich nutzen, um Klassen mit Eigenschaften auszustatten, die nicht entlang der Klassenhierarchie organisiert sind.

Beispiel 25.2-1: Verderbliche Waren in der Artikelhierarchie

Betrachten wir noch einmal die verschiedenen Artikelgruppen eines Blumenladens. Es liegt nahe, zunächst zwischen Pflanzen und Zubehör zu unterscheiden. Pflanzen können weiter in Topfpflanzen, Schnittblumen, Trockenblumen usw. aufgeteilt werden, Zubehör lässt sich beispielsweise in Gefäße, Pflegemittel, Schmuck usw. auffächern. Eine solche Hierarchie ist sinnvoll gewählt, weil durch Spezialisierung die relevanten Eigenschaften konkreter Artikel präzise abgebildet werden können. Nun kann es aber Eigenschaften geben, die an mehreren Stellen der Hierarchie relevant sind, jedoch nicht durchgehend in einem einzelnen Zweig auftreten.

Als Beispiel betrachten wir den Sachverhalt, dass gewisse Waren verderblich sind und deshalb bis zu einem festgelegten Zeitpunkt verkauft sein sollen, während andere Waren fast beliebig lange im Lagerbestand verbleiben können. Beliebig lange

haltbar sind Trockenblumen, Gefäße und Schmuck, während Topfpflanzen, Schnittblumen und Pflegemittel nur begrenzt haltbar sind. Um einen Lagerbestand

```
Artikel[] bestand;
```

nach verderblichen Waren abfragen zu können, benötigen wir zweierlei:

- *Wir müssen erkennen können, welche Artikel begrenzt haltbar sind.*
- *Wir wollen das Haltbarkeitsdatum dieser Artikel auf einheitliche Weise erfragen können.*

Zu diesem Zweck definieren wir eine Schnittstelle BegrenztHaltbar mit einer Methode liefereHaltbarkeitsdatum(). Klassen, die verderbliche Waren repräsentieren, implementieren die Schnittstelle BegrenztHaltbar und damit die Methode liefereHaltbarkeitsdatum(). Der Lagerbestand kann so dann wie folgt abgefragt werden:

```
for (Artikel artikel : bestand) {
    if (artikel instanceof BegrenztHaltbar) {
        // ...
        // verwende
        // ((BegrenztHaltbar) artikel).liefereHaltbarkeitsdatum()
    }
}
```

Abb. 25.2-1 zeigt die Artikelhierarchie mit der Schnittstelle BegrenztHaltbar. Beachten Sie, dass beliebig viele Klassen und Zweige der Artikelhierarchie die Schnittstelle implementieren können, während benachbarte Klassen und Zweige davon unberührt bleiben.

└

Selbsttestaufgabe 25.2-1:

Definieren Sie die Schnittstelle BegrenztHaltbar und schreiben Sie eine Klassendefinition für die Klasse Pflegemittel, die verderbliche Chemikalien repräsentiert. Der Rückgabotyp der Methode liefereHaltbarkeitsdatum() sei eine Klasse Datum, die Sie als leere Klassenvereinbarung implementieren können.

Bemerkung 25.2-1: Mehrfachvererbung und Vererbungskonflikte

Vererbungskonflikt

In Java tritt Mehrfachvererbung nur im Zusammenhang mit Schnittstellen auf. Von Schnittstellen erbt eine Klasse weder Exemplarvariablen noch Methodenimplementierungen.

Andere Programmiersprachen wie Common Lisp oder C++ kennen Mehrfachvererbung auch unter Klassen, was zur Folge haben kann, dass eine Unterklasse mehrere Exemplarvariablen a des gleichen Typs t oder mehrere unterschiedlich implementierte Methoden $m()$ mit der gleichen Signatur von verschiedenen Oberklassen erbt. Wir nennen dies einen Vererbungskonflikt, weil in solchen Fällen unklar ist,

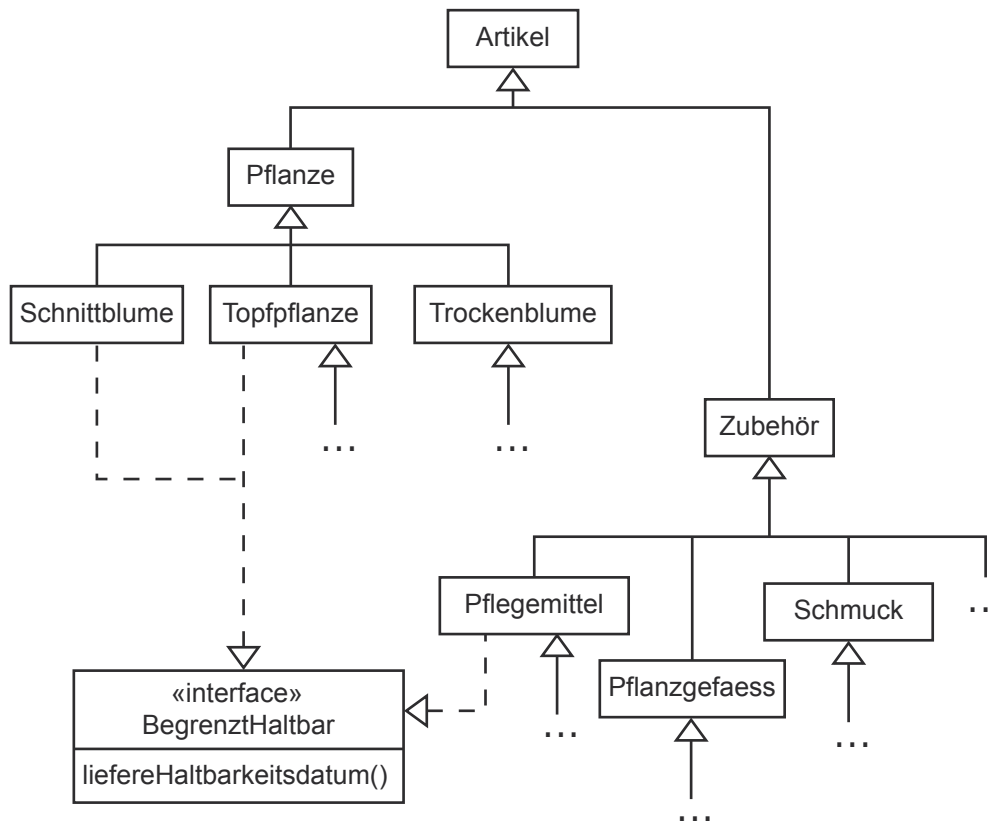


Abb. 25.2-1: Artikelhierarchie mit der Schnittstelle `BegrenztHaltbar`

auf welche Definition des Attributs sich ein Zugriff auf `a` bezieht oder welche Implementierung bei einem Aufruf von `m()` gemeint ist. Um solche Vererbungskonflikte zu vermeiden, gelten in Programmiersprachen mit Mehrfachvererbung entweder feste Regeln zur Auflösung von Vererbungskonflikten – etwa durch die Vorgabe einer Erbreihenfolge – oder ein Verbot von Vererbungskonflikten.

Falls in Java eine Klasse mehrere Schnittstellen implementiert, die Methoden mit gleicher Signatur definieren, so entsteht kein Vererbungskonflikt, denn es erfolgt nur eine einzige Methodenimplementierung in der Klasse selbst.

Treten Attribute gleichen Namens in verschiedenen Schnittstellen auf, die von einer Java-Klasse implementiert werden sollen, so müssen sie innerhalb der Klasse mit ihren qualifizierten Namen angesprochen werden, anderenfalls weist der Java-Übersetzer das Programm zurück [JLS: § 9.3.2.1]. Da es sich bei in Schnittstellen definierten Attributen immer um Klassenattribute handelt, ist ihr qualifizierter Name `Schnittstellename.Attributname`.

┘

26 Zeichenketten

Wir haben bereits in den vorhergehenden Kurseinheiten Zeichenketten verwendet. Nachdem uns inzwischen die konzeptionellen Grundlagen von Klassen und Objekten bekannt sind, wollen wir die Handhabung von Zeichenketten genauer betrachten.

Dieses Kapitel stellt zwei in Java eingebauten Klassen der Java-Laufzeitbibliothek zur Handhabung von Zeichenketten vor: `String` und `StringBuilder`. Der Unterschied zwischen den beiden Klassen besteht darin, dass `String`-Objekte Zeichenketten als Konstanten behandeln, während `StringBuilder`-Objekte veränderliche Zeichenketten verwalten.

Java-API-Referenz

Zugleich lernen wir die Dokumentation der Java-Laufzeitbibliothek kennen, die Java-API-Referenz¹⁶. Die Abkürzung API steht für Application Programming Interface. Die Java-API-Referenz können (und sollten) Sie immer dann konsultieren, wenn Sie sich einen Überblick über eine Klasse aus der Java-Laufzeitbibliothek verschaffen möchten. Die Java-API-Referenz ist vollständig in dem Sinne, dass sie zu jeder Klasse diejenigen Methoden und Attribute beschreibt, die außerhalb der Klassendefinition sichtbar und somit verwendbar sind. Später werden Sie lernen, wie Sie für Ihre selbst geschriebenen Klassen ähnliche Dokumentationen erstellen können.

26.1 Die Klasse `String`

`String`

Viele Programmiersprachen bieten einen primitiven Datentyp für Zeichenketten an, Java jedoch nicht. Stattdessen werden für Zeichenketten automatisch Objekte der Klasse `String` erzeugt. [JLS: § 4.3.3]

Definition 26.1-1: Zeichenketten in Java

Zeichenketten in Java

Eine Folge von Zeichen, die in Anführungsstrichen eingeschlossen sind, wie z. B.

`"Der höchste Berg Javas ist der Vulkan Semeru."`

erzeugt in Java automatisch ein Objekt der Klasse `String`, dessen Wert mit der Zeichenfolge übereinstimmt.

`String`-Literal

Ein auf diese Weise erzeugtes `String`-Objekt wird `String`-Literal genannt. [JLS: § 3.10.5]

┘

¹⁶ <http://java.sun.com/j2se/1.5.0/docs/api/>

Ein `String`-Literal kann einer Variablen von Typ `String` zugewiesen werden, woraus sich die Kurzform zur Deklaration von Zeichenketten erklärt:

```
String nachname = "Müller";
```

Zur Erzeugung von `String`-Objekten kennen wir weiterhin bereits den folgenden Konstruktor:

```
String vorname = new String("Heinz");
```

Dieser Konstruktor empfängt als Parameter ein `String`-Literal und liefert eine Kopie dieses `String`-Objekts.

Die Klasse `String` stellt viele weitere Konstruktoren bereit, die jeweils unterschiedliche Parametertypen erwarten, der Folgende dient beispielsweise zur Ableitung eines `String`-Objekts aus einem Feld von Zeichen:

```
char[] letters = {'H', 'a', 'g', 'e', 'n'};
String city = new String(letters);
```

Der Verkettungsoperator `+` ist eine weitere Besonderheit der Klasse `String`. Er erlaubt es uns, Zeichenketten durch Aneinanderreihung zu einer neuen Zeichenkette zu vereinen [JLS: § 15.18.1]. So entsteht durch die Verkettung

Verkettungsoperator `+`

```
"Java-" + "Kaffee"
```

eine neue Zeichenkette `"Java-Kaffee"`. Der Operator `+` ist uns bereits bei den zahlenwertigen Datentypen begegnet und wurde dort zur Addition numerischer Werte benutzt. Der Operator ist überladen und wird im Zusammenhang mit Zeichenketten mit einer weiteren Bedeutung versehen.

Falls ein Operand von `+` eine Zeichenkette ist und andere Operanden nicht, so werden die anderen Operanden in Zeichenketten überführt, bevor die Konkatenation ausgeführt wird. Dies kann insbesondere in Verbindung mit Zuweisungen oder Ausgabeanweisungen genutzt werden. So reiht die folgende Anweisung drei Zeichenketten aneinander und druckt sie aus:

Konkatenation

```
int glueckszahl;
...
System.out.println("Ihre persönliche Glückszahl lautet "
    + glueckszahl + ".");
```

Wenn `glueckszahl` den Wert 17 aufweist, erscheint die Ausgabe:

```
Ihre persönliche Glückszahl lautet 17.
```

Wird ein `String`-Objekt mit dem Verkettungsoperator mit einem Objektverweis verknüpft, so wird automatisch an diesem Objekt die `toString()`-Methode aufgerufen und so eine passende Zeichenkettenrepräsentation erzeugt. [JLS: § 15.18.1.1]

Automatische
Umwandlung in eine
Zeichenkette

Selbsttestaufgabe 26.1-1:

Welchen Wert haben die folgenden Ausdrücke?

- "413" + 24
- "08" + 10 + 5
- "08" + (10 + 5)



Die Klasse `String` stellt eine große Anzahl von Methoden bereit, von denen hier eine Auswahl vorgestellt wird. In der Java-API-Referenz der Klasse finden sich über 50 Methoden.

Ein `String`-Objekt kann auf vielfältige Weise über sich selbst Auskunft geben. So erhalten wir mit

```
int laenge = vorname.length(); // liefert 5
```

die Anzahl der in der Zeichenkette enthaltenen Zeichen. Die Methode

```
int ePosition = vorname.indexOf('e'); // liefert 1
```

sucht in einer Zeichenkette nach dem ersten Auftreten des Zeichens 'e' und gibt den Indexwert des Zeichens zurück. Die Methode `indexOf()` ist überladen und akzeptiert als Parameter auch Zeichenketten. So sucht

```
int inPosition = vorname.indexOf("in"); // liefert 2
```

in einer Zeichenkette nach dem ersten Auftreten der Teilzeichenkette "in". Der Rückgabewert ist der Indexwert des Anfangs der Fundstelle. Beachten Sie, dass die Zählung der Zeichen bei 0 beginnt. Nachdem `vorname` mit der Zeichenkette "Heinz" belegt wurde, ergibt `vorname.indexOf('e')` den Wert 1 und `vorname.indexOf("in")` den Wert 2. Das Ergebnis -1 zeigt an, dass ein Zeichen oder eine Zeichenkette nicht gefunden wurde.

Eine gewissermaßen reziproke Auskunft, nämlich ein Zeichen an einem bestimmten Indexwert der Zeichenkette, erhalten wir mit folgender Methode:

```
char viertesZeichen = vorname.charAt(3); // liefert 'n'
```

Die `indexOf()`-Methoden lassen sich auch nutzen, um das Vorkommen eines bestimmten Zeichens oder einer Zeichenkette zu prüfen:

```
if (vorname.indexOf('o') == -1) {
    System.out.println("In " + vorname
        + " kommt kein 'o' vor.");
}
```

Komfortabler können wir eine solche Prüfung mit der `contains()`-Methode realisieren, die als Parameter u. a. ein `String`- oder ein `StringBuilder`-Objekt entgegen nimmt und einen Wahrheitswert als Ergebnis liefert:

```
boolean eiKommtVor = vorname.contains("ei");
```


Selbsttestaufgabe 26.1-2:

Sei

```
String s = "Hallo Welt";
```

Was liefert der folgende Ausdruck:

```
s.indexOf('l') + s.lastIndexOf('l')
```

Lösungshinweis: Die Methode `lastIndexOf()` wurde oben nicht vorgestellt. Ihr sprechender Name sollte Sie nicht davon abhalten, die Java-API-Referenz zu konsultieren.



Selbsttestaufgabe 26.1-3:

Wie prüfen Sie in einer if-Anweisung, ob ein String s mit der Zeichenkette "Morgenstund" beginnt und mit der Zeichenkette "Mund." endet?

Lösungshinweis: Der Test ließe sich durch eine geschickte Kombination einer `indexOf()`-Methode und der `length()`-Methode formulieren; verwenden Sie stattdessen die Java-API-Referenz und setzen Sie geeignetere Methoden ein.



Selbsttestaufgabe 26.1-4:

Gibt es Test-Zeichenketten, mit denen im folgenden Programm-Ausschnitt der else-Zweig erreicht wird? Begründen Sie Ihre Antwort!

```
String test = new String(...); // Hier die Test-Zeichenkette
                               // einsetzen

int position = 3;
if (test.indexOf(test.charAt(position)) == position) {
    System.out.println("Kein Wunder!");
} else {
    System.out.println("Ach!");
}
```



Neben Methoden zur Untersuchung der Zeichenketten enthält die Klasse `String` auch Methoden, um Zeichenketten miteinander zu vergleichen. Dabei fordern wir jeweils ein `String`-Objekt auf, sich mit einem anderen `String`-Objekt zu vergleichen. Seien

```
String s = "neben";
String t = "einander";
String u = "nebeneinander";
String w = s + t;
```

Die Methode `equals()` liefert `true`, falls zwei Zeichenketten lexikalisch gleich sind, ansonsten `false`:

```
boolean istGleich;
istGleich = s.equals(t);      // liefert false
istGleich = u.equals(s + t); // liefert true
istGleich = u.equals(w);      // liefert true
```

Bemerkung 26.1-1:

Die Methode `equals()` ist uns als Methode der Klasse `Object` bereits begegnet (vgl. Abschnitt 22.5). Auch die Klasse `String` ist eine Subklasse von `Object`; durch Überschreiben der `equals()`-Methode definiert sie die Gleichheit unter `String`-Objekten um.

┘

Mit der Methode `compareTo()` erhalten wir ein differenzierteres Ergebnis. Sie liefert eine Ganzzahl, wobei drei Fälle auftreten können:

- Das Ergebnis 0 bedeutet, dass die beiden Zeichenketten lexikalisch gleich sind (und entspricht somit dem Fall, dass `equals()` `true` liefert).
- Eine negative Ganzzahl bedeutet, dass die Zeichenketten des `String`-Objekts, dessen `compareTo()`-Methode aufgerufen wurde, alphabetisch vor der Zeichenkette des Parameters steht.
- Eine positive Ganzzahl signalisiert die umgekehrte Reihenfolge.

```
int vergleich;
vergleich = u.compareTo(w); // liefert 0
vergleich = t.compareTo(s); // liefert -9
vergleich = s.compareTo(t); // liefert 9
```

Bemerkung 26.1-2:

Gelegentlich werden Ihnen Vergleiche der folgenden Art begegnen:

```
if (u == w) { ... }
```

`u` und `w` sind die Namen von `String`-Objekten und stellen somit Referenzen auf diese Objekte dar. Der Vergleichsoperator `==` prüft, ob `u` und `w` auf dasselbe Objekt verweisen. Das Ergebnis dieses Vergleichs ist hier `false` – unabhängig davon, dass `u.equals(w)` `true` ergibt.

In gewissen Fällen kann ein Vergleich von `String`-Objekten mit dem Vergleichsoperator `==` auch zu überraschenden Ergebnissen führen:

```
boolean erstaunlich = (s == "neben"); // liefert true
```

Dieser Effekt ist dadurch zu erklären, dass die Laufzeitumgebung mit einer internen Optimierung arbeitet und versucht, die Aufbewahrung mehrerer gleicher Zeichenketten im Speicher zu vermeiden. In diesem Fall hat sie für "neben" gar nicht erst ein neues `String`-Objekt erzeugt, sondern lediglich eine interne Referenz angelegt, die auf das bereits im Speicher existierende `String`-Objekt mit der gleichen Zeichenkette "neben" verweist.

Von der Verwendung des Vergleichsoperators == zum lexikalischen Vergleich wird abgeraten. Ob die Laufzeitumgebung interne Referenzen oder „wahrhaftige“ String-Objekte anlegt, hängt von verschiedenen (nicht immer überschaubaren) Faktoren ab.

┘

Selbsttestaufgabe 26.1-5:

Schreiben Sie eine Methode `first(String a, String b, String c)`, die von drei gegebenen Zeichenketten `a`, `b` und `c` diejenige ausgibt, die alphabetisch vor den beiden anderen steht.

Wie reagiert Ihre Methode bezüglich Groß- und Kleinschreibung? Welche Methode aus der Java-API-Referenz verändert das Verhalten bezüglich Groß- und Kleinschreibung?

◇

Wir können ein `String`-Objekt auch dazu veranlassen, neue `String`-Objekte zu produzieren. So ergibt

```
String vollerName = vorname.concat(nachname);
```

die (etwas unschöne) Aneinanderreihung `"HeinzMüller"` als Zeichenkette eines neuen `String`-Objekts `vollerName`. Die beiden `String`-Objekte mit den Zeichenketten `"Heinz"` und `"Müller"` bleiben unverändert. Auch alle folgenden Methoden lassen das `String`-Objekt `vorname` unverändert. Mit

```
String neuerName = vorname.replace('z', 'o');
```

erhalten wir für `neuerName` eine Zeichenkette, in der alle Vorkommen des Zeichens `'z'` durch das Zeichen `'o'` ersetzt sind. Weiterhin können wir Teilzeichenketten und Zeichenketten aus Klein- oder Großbuchstaben produzieren lassen:

```
String ausschnitt = vorname.substring(1, 4) // ergibt "ein"
```

```
String allesklein = vorname.toLowerCase() // ergibt "heinz"
```

```
String allesgross = vorname.toUpperCase() // ergibt "HEINZ"
```

Beachten Sie: Da `String`-Objekte unveränderlich sind, liefern alle letztgenannten Methoden neue `String`-Objekte.

Selbsttestaufgabe 26.1-6:

Welche Ausgabe erzeugen die folgenden Anweisungen:

```
String a = "Das hier ist ganz viel Text.";
String b = "awerhgarsdasfasnichtopjkfasf";
System.out.println(a.substring(0,13) + b.substring(5,8) + " "
    + b.substring(15,20).toUpperCase()
    + a.substring(17));
```

◇

Selbsttestaufgabe 26.1-7:

Im Datenlexikon in Abschnitt 2.7 hatten wir Namen als eine Folge von Buchstaben, die auch Bindestriche enthalten kann, aber mit einem Buchstaben beginnen muss, vorgegeben.

Schreiben Sie eine Methode `boolean checkName(String name)`, die für eine gegebene Zeichenkette `name` überprüft, ob sie den o. g. Namenskonventionen genügt. Bei der Eingabe akzeptieren wir Klein- und Großbuchstaben. Die Methode soll zudem einen gültigen Namen auf der Konsole so ausdrucken, dass der erste Buchstabe groß und der Rest klein geschrieben wird.

Zur Vereinfachung wollen wir es zulassen, dass ein Name mehrere Bindestriche hintereinander enthalten darf. Umlaute und andere Sonderzeichen wie 'ß' vernachlässigen wir.



26.2 Die Klasse `StringBuilder`

`StringBuilder`

Die Benutzung von `StringBuilder`-Objekten ist immer dann angebracht, wenn wir zahlreiche Operationen auf Zeichenketten vornehmen, Zeichenketten beispielsweise verkürzen oder verlängern, Teile einer Zeichenkette miteinander vertauschen oder eine Teilzeichenkette durch eine andere ersetzen.

Wir könnten dies prinzipiell auch mit `String`-Objekten tun, jedoch würden mit jeder Manipulation neue `String`-Objekte erzeugt werden. Die Klassen `StringBuilder` arbeitet zum einen effizienter, zum anderen stellt sie Methoden bereit, die die Manipulation von Zeichenketten komfortabel gestalten. Wir betrachten wiederum eine kleine Auswahl.

Zur Erzeugung von `StringBuilder`-Objekten stehen unter anderem die folgenden drei Konstruktoren zur Auswahl:

```
StringBuilder littleBuilder = new StringBuilder();
```

erzeugt ein `StringBuilder`-Objekt mit einer Start-Kapazität von 16 Zeichen, das noch keine Zeichen enthält.

```
StringBuilder bigBuilder = new StringBuilder(1000);
```

erzeugt ein `StringBuilder`-Objekt mit der angegebenen Start-Kapazität, das noch keine Zeichen enthält.

```
StringBuilder sammelBuilder = new StringBuilder("Ein Anfang");
```

erzeugt ein `StringBuilder`-Objekt, das ab der ersten Indexposition die angegebene Zeichenkette enthält. Die Start-Kapazität entspricht der Länge der Zeichenkette zuzüglich 16 Zeichen.

Um Zeichenketten zu manipulieren, können wir mit

```
sammelBuilder.append(", aber jetzt geht es weiter");
```

Zeichenketten anhängen. Dabei verwalten Objekte der Klasse `StringBuilder` ihre Kapazität und vergrößern diese automatisch, wenn es erforderlich ist. Mit

```
sammelBuilder.insert(30, " ein wenig");
```

können wir Zeichenketten an einer gewünschten Stelle einfügen und mit

```
sammelBuilder.delete(12, 17);
```

Teile der Zeichenkette entfernen. Die Zählung der Indexpositionen beginnt bei 0, wobei sowohl beim Einfügen als auch beim Löschen der imaginäre Zwischenraum vor dem jeweiligen Zeichen gemeint ist. Mit `insert(3, "etwas")` fügen wir die neue Zeichenkette zwischen den Indexpositionen 2 und 3 ein, d. h. zwischen dem dritten und vierten Zeichen. Mit `delete(3, 5)` entfernen wir die Zeichen mit den Indexpositionen 3 und 4, d. h. das vierte und fünfte Zeichen.

Die Methoden `append()` und `insert()` sind überladen und akzeptieren viele Datentypen als Parameter.

Beachten Sie: Die genannten Methoden verändern direkt die Zeichenkette des aufgerufenen `StringBuilder`-Objekts. Zugleich liefern sie einen Rückgabewert, und zwar eine Referenz auf dasselbe `StringBuilder`-Objekt (also auf sich selbst). Der Rückgabewert findet selten Verwendung. Daraus folgt, dass nach der Ausführung von

```
StringBuilder vorname = new StringBuilder("Heinz");
StringBuilder vollerName = vorname.append("Müller");
```

beide Variablen auf dasselbe `StringBuilder`-Objekt verweisen, das die Zeichenkette "HeinzMüller" enthält.

Bemerkung 26.2-1:

Die Zeichenkette eines `StringBuilder`-Objekts kann ebenso wie diejenige eines `String`-Objekts mit der folgenden Anweisung ausgegeben werden:

```
System.out.println(sammelBuilder);
```

┘

Selbsttestaufgabe 26.2-1:

Manipulieren Sie das folgende `StringBuilder`-Objekt so, dass es im Ergebnis die Zeichenkette "String-Objekte sind unveränderlich." enthält.

```
StringBuilder sb = new StringBuilder(
    "StringBuilder-Objekte sind veränderlich.");
```


◇

Selbsttestaufgabe 26.2-2:

Manipulieren Sie das folgende `StringBuilder`-Objekt so, dass es im Ergebnis die Zeichenkette "NOTREGAL" enthält. Konsultieren Sie die Java-API-Referenz.

```
StringBuilder sb = new StringBuilder("LAGERTONNE");
```

**Selbsttestaufgabe 26.2-3:**

Ergänzen Sie die Klassen `Artikel` und `Pflanze` aus Abschnitt 22.4 so, dass sie die Methode `toString()` der Klasse `Object` überschreiben. Die zurückgelieferte Zeichenkette soll dieselben Daten enthalten, wie sie durch die jeweiligen `gebeInformationenAus()`-Methoden am Bildschirm ausgegeben werden. 

27 Zusammenfassung

Vererbung ist ein Schlüsselmechanismus objektorientierter Programmiersprachen. Vererbung etabliert eine **ist-ein-Beziehung** zwischen einer Unter- und ihrer Oberklasse. Die Rolle der **Unterklasse** besteht darin, das von den **Oberklassen** geerbte Verhalten und die geerbten Eigenschaften und Verhalten zu erweitern und das Verhalten (durch Überschreiben) zu verändern.

Bei der Erweiterung wird das nach außen sichtbare Verhalten der Objekte der Unterklasse um die Möglichkeit erweitert, auf neue Botschaften zu reagieren. Beim **Überschreiben** wird die Implementierung einer Methode und damit die Art und Weise, wie eine Methode Aufrufe behandelt, verändert. Dies geschieht, indem die Unterklasse eine Methode mit der gleichen Signatur implementiert. Dabei darf die Sichtbarkeit nur erweitert, nicht jedoch eingeschränkt, werden. Überschreiben von Methoden darf nicht mit dem Überladen von Methodennamen, wo mehrere Methoden mit dem gleichen Namen, aber verschiedenen Signaturen implementiert werden, verwechselt werden. **Finale Methoden** können nicht überschrieben werden.

Auf überschriebene Methoden und **verdeckte Attribute** der Oberklasse kann mit Hilfe des Schlüsselwortes `super` zugegriffen werden.

Java unterstützt die **Einfachvererbung**, bei der jede Klasse immer nur eine Oberklasse erweitert. Die Oberklasse wird in Java durch das Schlüsselwort `extends` in der Klassendeklaration angegeben. Auch wenn eine Java-Klasse nicht ausdrücklich eine andere Klasse erweitert, so erweitert sie implizit die Klasse `Object`, die von Java bereitgestellt wird. Durch diese Festlegung ist jede Java-Klasse ein Abkömmling der Klasse `Object`. Von **finalen Klassen** kann es keine Unterklassen geben.

Klassen können auf der Grundlage anderer Klassen aufbauen und müssen nicht komplett neu entwickelt werden. Vererbung unterstützt somit die **systematische Wiederverwendung** von Programmcode auf strukturierte Weise und erleichtert die Programmanpassung an neue Erfordernisse.

Die **Schnittstellenvererbung** ermöglicht die Wiederverwendung von Funktionalität aber nicht von Implementierungen. Zudem kann so Funktionalität, die sich orthogonal zur Klassenhierarchie verhält, definiert werden. Sie bestimmt, wann ein Objekt an Stelle eines Objekts anderen Typs verwendet werden kann. Objekte mit gleicher Schnittstelle können aber verschiedene Implementierungen haben. Schnittstellen unterstützen außerdem eine eingeschränkte Form der **Mehrfachvererbung**, die das Problem von Vererbungskonflikten bei Methoden ausschließt. Eine Klasse kann eine oder mehrere Schnittstellen durch deren Angabe in Verbindung mit dem Schlüsselwort `implements` in der Klassendeklaration implementieren.

Abstrakte Klassen hingegen, die durch das Schlüsselwort `abstract` gekennzeichnet werden, ermöglichen durchaus die Angabe von Implementierungen und

Attributen, jedoch können sie auch für einzelne Methoden keine Implementierung angeben. Diese müssen dann von den Unterklassen implementiert werden. Von einer abstrakten Klasse kann es keine Exemplare geben.

Parallel zur Klassenhierarchie entsteht auch eine **Typhierarchie**. Der Typ einer Unterklasse ist immer auch ein **Subtyp** des Oberklassentyps. Das gleiche gilt auch zwischen den von Schnittstellen definierten Typen. Ein Typ einer Klasse ist auch immer ein Subtyp aller Typen der Schnittstellen, die von der Klasse implementiert werden.

Das **Substitutionsprinzip** ermöglicht es, Objekte von Subtypen dort zu verwenden, wo Objekte des allgemeineren Typs erwartet werden. Deshalb kann bei Verweisvariablen und Ausdrücken zwischen dem **statischen und dynamischen Typ** unterschieden werden. In Kombination mit dem Überschreiben von Methoden kann so **vielgestaltiges Verhalten (Polymorphie)** entstehen, da verschiedene Unterklassen auf die gleiche Nachricht mit verschiedenem Verhalten reagieren. Die Entscheidung, welche Methode genau ausgeführt wird, wird bei Exemplarmethoden erst zur Laufzeit getroffen. Deshalb spricht man auch von **dynamischer Bindung**. Bei Attributen und Klassenmethoden wird die Entscheidung schon bei der Übersetzung getroffen, weshalb man von **statischer Bindung** spricht.

Ein **Paket** ist eine Sammlung inhaltlich zusammengehöriger Klassen und Schnittstellen, die den Zugriffsschutz und die Verwaltung von Namensräumen unterstützt. Pakete erleichtern es, Klassen zu finden und zu nutzen, und helfen dabei, Namenskonflikte zu verhindern.

Die Sichtbarkeit von Klassen und ihren Elementen sowie der Zugriff auf sie kann mit Hilfe von **Zugriffsmodifikatoren** kontrolliert werden.

Dabei sind **öffentliche Klassen** durch den Modifikator `public` gekennzeichnet, und sie sind in allen Paketen sichtbar. Klassen ohne Modifikator besitzen die **Standardsichtbarkeit** und sind nur im eigenen Paket sichtbar.

Öffentliche Klassenelemente sind überall dort sichtbar, wo ihre Klasse sichtbar ist.

Private Klassenelemente, die durch den Modifikator `private` gekennzeichnet sind, sind nur innerhalb der Klasse, und Elemente ohne Modifikator, also mit **Standardsichtbarkeit** nur innerhalb des Pakets der Klasse sichtbar.

Geschützte (protected) Elemente sind innerhalb des Pakets der Klasse und in Unterklassen, unabhängig von deren Paket, verfügbar. Aber nur dann, wenn dort auf ein Objekt der Unterklasse zugegriffen wird.

Zeichenketten (Objekte der Klasse `String`) sind Objekttypen in Java, und als solche verhalten sie sich auch wie Objekte. Java unterstützt einige vereinfachte Schreibweisen für Zeichenketten sowie vordefinierte Operatoren, mit deren Hilfe man mit `String`-Objekten ähnlich wie mit elementaren Datentypen umgehen kann. Bei der **Bearbeitung von Zeichenketten** sollte hingegen die Klasse `StringBuilder` verwendet werden, da diese mit veränderlichen Zeichenketten besser umgehen kann.

Lösungen zu Selbsttestaufgaben der Kurseinheit

Lösung zu Selbsttestaufgabe 22.2-1:

`new Student().gebeMatrikelnrAus()` ist korrekt

`new Mitarbeiter().abteilung.inhaber` erzeugt einen Übersetzungsfehler, da Mitarbeiter auf eine Organisationseinheit verweisen und nur Lehrgebiete einen Inhaber haben, aber nicht jede Organisationseinheit

`new Person().gebeNameAus()` ist korrekt

`new Professor().gebeRaumnummerAus()` ist korrekt, da die Methode geerbt wird

`new Student().gebeNameAus()` ist korrekt, da die Methode geerbt wird

`new Professor().abteilung.name` ist korrekt

`new Mitarbeiter().gebeMatrikelnrAus()` erzeugt einen Übersetzungsfehler, da die Klasse Mitarbeiter keine Unterklasse von Student ist

`new Lehrgebiet().inhaber.gebeRaumnummerAus()` ist korrekt

`new Person().getRaumnummerAus()` erzeugt einen Übersetzungsfehler

Lösung zu Selbsttestaufgabe 22.2-2:

Student ist Subtyp von Person

Mitarbeiter ist Subtyp von Person

Professor ist Subtyp von Person

Professor ist Subtyp von Mitarbeiter

Lehrgebiet ist Subtyp von Organisationseinheit

Lösung zu Selbsttestaufgabe 22.2-3:

```
class Person {
    String name;
}
```

```
class Kunde extends Person {}
```

```

class Premiumkunde extends Kunde {
    long kundennummer;
    String adresse;
}

class Angestellter extends Person {
    String adresse;
    double gehalt;
    long sozialversicherungsnummer;
}

```

Lösung zu Selbsttestaufgabe 22.3-1:

Tab. ML 16: Statischer und dynamischer Typ

	statischer Typ	dynamischer Typ
x	Artikel	Pflanze
y	Pflanze	Pflanze
z	Artikel	Artikel

Lösung zu Selbsttestaufgabe 22.3-2:

Tab. ML 16: Statischer und dynamischer Typ

	statischer Typ	dynamischer Typ
new Artikel()	Artikel	Artikel
X.doSomething(null)	Artikel	Pflanze
((Artikel) new Pflanze())	Artikel	Pflanze
X.doSomething(new Artikel())	Artikel	Artikel
((Pflanze) X.doSomething(null))	Pflanze	Pflanze

Lösung zu Selbsttestaufgabe 22.3-3:

`C t = (C) s;` erzeugt eine `ClassCastException`, da ein Verweis auf ein Objekt vom Typ `A` nicht auf einen Verweis vom Typ `C` geändert werden kann

`new D().k(new E());` erzeugt einen Übersetzungsfehler, da `E` kein Subtyp von `B` ist, was aber als Argument erwartet wird.

`new C().g(s).k(null);` erzeugt einen Laufzeitfehler, da in der Methode `k()` versucht wird, auf ein Attribut an einem `null`-Verweis zuzugreifen.

Lösung zu Selbsttestaufgabe 22.4-1:

Das Überschreiben von Methoden erfordert es, dass die neue Methode die gleiche Signatur erhält, wie die überschriebene Methode der Oberklasse. Wenn die Signaturen der geerbten und der neuen Methode unterschiedlich sind, wird die neue Methode die geerbte Methode überladen aber nicht überschreiben. Überschreiben von Methoden kann nur in einer Unterklasse erfolgen. Überladen von Methodennamen ist jedoch auch innerhalb einer einzigen Klasse möglich.

Lösung zu Selbsttestaufgabe 22.4-2:

Es wird die folgende Ausgabe erzeugt:

```
11
9
18
22
23
17
```

Lösung zu Selbsttestaufgabe 22.4-3:

Die Methode `a()` in der Klasse `Unter` hat keinen gültigen Ergebnistyp, denn dieser muss beim Überschreiben ein Subtyp des in der Oberklasse deklarierten Ergebnistyps sein.

Für die Methode `a()` in `Unter` wären `Unter` und `UnterUnter` zulässige Ergebnistypen. Die Veränderung des Ergebnistyps bei der Methode `b()` in der Klasse `UnterUnter` ist deswegen zulässig.

Lösung zu Selbsttestaufgabe 22.4-4:

```
10
10
10
3
2
3
10
2
```

Lösung zu Selbsttestaufgabe 22.7-1:

Die Suche beginnt zur Übersetzungszeit beim statischen Typ C. Die Klasse C definiert selbst keine passende Methode. In der Klasse B wird dann mit `z(Object)` eine mögliche passende Methode gefunden. Da der statische Typ (`String`) des Arguments "Hallo" nicht mit dem Typ des Parameters übereinstimmt, wird die Hierarchie noch weiter durchsucht. In der Klasse A wird dann die Methode `z(String)` gefunden, und da der Typ des Arguments mit dem Parameter-typ übereinstimmt, kann die Suche beendet werden. Die Signatur lautet folglich: `z(String)`.

Da es sich um eine Exemplarmethode handelt, wird zur Laufzeit beim dynamischen Typ E mit der Suche nach der Methode begonnen. In E selber ist jedoch keine Methode mit der passenden Signatur definiert. In der Klasse D ist jedoch eine Methode mit der Signatur `z(String)` definiert. Somit ist die Methode gefunden, die zur Laufzeit ausgeführt wird. Es muss nicht mehr weiter gesucht werden, da immer die Methode ausgeführt wird, die am weitesten unten in der Klassenhierarchie zu finden ist.

Es wird `D.z(String)` ausgeführt und die Ausgabe "D.z" erzeugt.

Lösung zu Selbsttestaufgabe 22.7-2:

Beim ersten Aufruf wird `ZZZ.a(Object)` ausgeführt, da zur Übersetzungszeit bei der Suche nach einer passenden Methode beim statischen Typ, also `ZZZ`, angefangen wird. Dort wird dann die Signatur `a(Object)` gefunden und da die Methode `a(String)` in `YYY` diese nicht überschreibt, wird zur Laufzeit auch `ZZZ.a(Object)` ausgeführt.

Beim zweiten Aufruf wird `YYY.a(String)` ausgeführt, da zur Übersetzungszeit bei der Suche nach einer passenden Methode beim statischen Typ, also `YYY`, angefangen wird.

Lösung zu Selbsttestaufgabe 23.2-1:

Der Programmtext ist nicht korrekt. Klasse A wurde nicht für die Klasse d.D importiert. Das gleiche gilt für Klasse C, da das Platzhaltersymbol „*“ nur Klassen eines Paketes und keine Unterpakete importiert. Die Deklaration von Klasse d.D müsste also lauten:

```
package d;
import a.A;
import b.c.C;
class D {
    A myA = new A();
    C myC = new C();
}
```

oder:

```
package d;
class D {
    a.A myA = new a.A();
    b.c.C myC = new b.c.C();
}
```

Lösung zu Selbsttestaufgabe 24.3-1:

Tab. ML 16: Zugriffsmodifikator von f

	public	protected	(ohne)	private
Z1	+	+	-	-
Z2	+	+	+	+
Z3	+	+	+	-
Z4	-	-	-	-
Z5	+	-	-	-
Z6	+	+	-	-
Z7	+	+	-	-
Z8	+	-	-	-
Z9	+	-	-	-
Z10	-	-	-	-

+ Zugriff möglich

- Zugriff nicht möglich

Zugriff Z4 und Z10 sind niemals möglich, da B und D kein Attribut `f` besitzen und auch keines erben.

Lösung zu Selbsttestaufgabe 24.3-2:

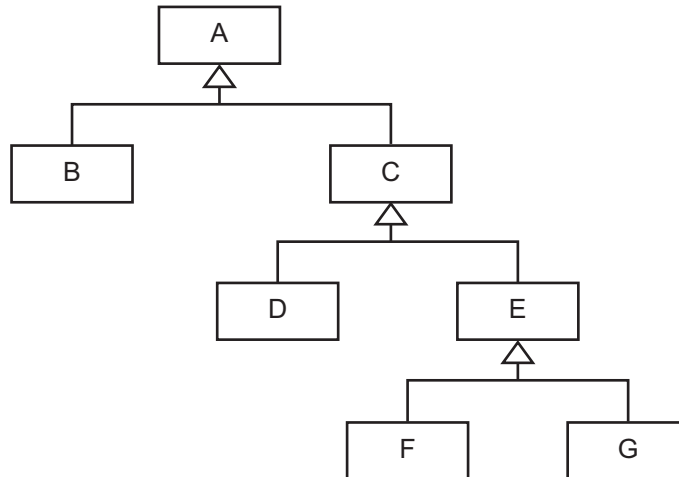


Abb. ML 17: Klassenhierarchie

Die folgenden Zeilen sind fehlerhaft:

- 9: Die Methode `C.n(int)` ist `private` und wird somit nicht von G geerbt.
- 12: Die Klasse G besitzt oder erbt lediglich Methoden mit den folgenden Signaturen: `m(String)`, `m(String, String)`, `m(int, String)`.
- 14: Das gleiche gilt für die Klasse E.
- 15: Die Methode `C.n(int)` ist als `private` deklariert und somit außerhalb der Klasse C nicht sichtbar.

Nach dem Löschen oder Auskommentieren dieser vier Zeilen erhält man bei der Ausführung die folgende Ausgabe:

```

G: hello
E: hello
E: true
C: hello guys
A: 2 times
F: 100
A: 4 you
  
```

Lösung zu Selbsttestaufgabe 25.1-1:

Das Programm ist fehlerhaft. Wenn eine Klasse, in unserem Fall A, abstrakte Methoden beinhaltet, so muss sie selbst abstrakt sein. Die Methode `fourTimes(int i)` darf nicht als abstrakt deklariert sein, da sie eine Implementierung aufweist. Die abstrakte Methode `twoTimes()` darf jedoch aufgerufen werden.

Lösung zu Selbsttestaufgabe 25.1-2:

Die Klasse Zylinder lässt sich komfortabel durch Spezialisierung der Klasse Topf implementieren:

```
public class Zylinder extends Topf {

    public Zylinder (double hoehe, double durchmesser) {
        super(hoehe, durchmesser, durchmesser);
    }
}
```

Die Klasse Wanne kann wie folgt implementiert werden:

```
public class Wanne extends Pflanzgefaess {

    private double laenge;

    public Wanne(double hoehe, double laenge) {
        this.legeHoeheFest(hoehe);
        this.legeLaengeFest(laenge);
    }

    public double berechneVolumen() {
        return this.liefereLaenge()
            * this.liefereHoehe()
            * this.liefereHoehe() * PI / 2;
    }

    // ... (weitere Methoden, insb. Setter- und
    //      Getter-Methoden für laenge)
}
```

Lösung zu Selbsttestaufgabe 25.1-3:

```
public abstract class Konto {

    protected double kontostand = 0;

    public Konto(final double betrag) {
        this.einzahlen(betrag);
    }

    public void einzahlen(final double betrag) {
        if (betrag < 0) {
            System.out.print("Negative Beträge können nicht ");
            System.out.println("eingezahlt werden.");
        } else {
            this.kontostand += betrag;
        }
    }

    abstract public boolean abheben(final double betrag);
}

public class Sparkonto extends Konto {

    public Sparkonto(final double betrag) {
        super(betrag);
    }

    public boolean abheben(final double betrag) {
        if (betrag > this.kontostand) {
            System.out.println("Sie können maximal "
                               + this.kontostand + " abheben.");
            return false;
        } else {
            this.kontostand -= betrag;
            return true;
        }
    }
}

public class Girokonto extends Konto {

    private double dispo = 0;

    public Girokonto(final double betrag) {
        super(betrag);
    }

    public void legeDispoFest(final double limit) {
        this.dispo = limit;
    }
}
```



```

public boolean abheben(final double betrag) {
    if (betrag > this.kontostand + this.dispo) {
        System.out.println("Sie können maximal "
            + (this.kontostand + this.dispo)
            + " abheben.");
        return false;
    } else {
        this.kontostand -= betrag;
        return true;
    }
}
}

```

Lösung zu Selbsttestaufgabe 25.2-1:

```

public interface BegrenztHaltbar {

    Datum liefereHaltbarkeitsdatum();
}

public class Pflegemittel extends Zubehoer
    implements BegrenztHaltbar {

    private Datum haltbarkeitsdatum;
    // ... (weitere Attribute)

    public Pflegemittel(final Datum haltbarBis) {
        this.legeHaltbarkeitsdatumFest(haltbarBis);
    }

    public Datum liefereHaltbarkeitsdatum() {
        return this.haltbarkeitsdatum;
    }

    // ... (weitere Methoden, insb. legeHaltbarkeitsdatumFest())
}

public class Datum {}

```

Lösung zu Selbsttestaufgabe 26.1-1:

- "413" + 24 ergibt "41324"
- "08" + 10 + 5 ergibt "08105"
- "08" + (10 + 5) ergibt "0815", weil nach den Vorrangregeln von Ausdrücken zunächst die Addition des int-Ausdrucks in Klammern und erst anschließend die Konkatenation ausgeführt wird.

Lösung zu Selbsttestaufgabe 26.1-2:

Der Ausdruck liefert als Ergebnis die Ganzzahl 10.

Lösung zu Selbsttestaufgabe 26.1-3:

```
if (s.startsWith("Morgenstund") && s.endsWith("Mund.")) {  
    ...  
}
```

Lösung zu Selbsttestaufgabe 26.1-4:

Die Antwort lautet: ja.

Der `else`-Zweig wird erreicht, wenn eine Test-Zeichenkette dasselbe Zeichen, das an vierter Stelle vorkommt, bereits an einer vorherigen Stelle enthält, wie beispielsweise in "Müller". Hier liefert `test.charAt(3)` das vierte Zeichen 'l', aber `test.indexOf('l')` wird bereits an der dritten Stelle fündig, liefert also den Wert 2.

Wenn die Test-Zeichenkette zu kurz ist, wird der `else`-Zweig hingegen nicht erreicht (wie man vielleicht angenommen hätte). Im Falle von

```
test.length() <= position
```

signalisiert Java bereits bei der Abarbeitung von `test.charAt(position)` eine Ausnahmesituation und liefert eine Fehlermeldung der Art `StringIndexOutOfBoundsException`.

Lösung zu Selbsttestaufgabe 26.1-5:

```
public void first(String a, String b, String c) {  
    System.out.println(first(a, (first(b, c))));  
}  
  
private String first(String a, String b) {  
    if ( a.compareTo(b) < 0 ) {  
        return a;  
    }  
    return b;  
}
```

Mit der Methode `compareTo()` werden alle Großbuchstaben vor allen Kleinbuchstaben eingeordnet. Die Methode `compareToIgnoreCase()` behandelt Groß- und Kleinbuchstaben als gleichwertig.

Lösung zu Selbsttestaufgabe 26.1-6:

Sie können die Ausgabe überprüfen, indem sie die drei Anweisungen in eine `main()`-Methode kleiden.

Lösung zu Selbsttestaufgabe 26.1-7:

Vergleichen Sie Ihre Lösung mit nachstehendem Programm. Falls Sie keine eigene Lösung gefunden haben, studieren und kommentieren Sie das folgende Programm ausführlich.

```
public class CustomerNames {

    public boolean checkName(String name) {
        if (!isEmptyString(name) &&
            firstIsChar(name) &&
            restIsNameChar(name)) {
            printName (name);
            return true;
        } else {
            System.out.println("Illegal name!");
            return false;
        }
    }

    private void printName(String name) {
        System.out.print (name.substring(0,1).
            toUpperCase());
        System.out.println(name.
            substring(1,name.length()).toLowerCase());
    }

    private boolean isLetter(char c) {
        return (c >= 'A' && c <= 'Z') ||
            (c >= 'a' && (c <= 'z'));
    }

    private boolean isNameChar(char c) {
        return isLetter(c) || isHyphen(c);
    }

    private boolean isHyphen(char c) {
        return c == '-';
    }

    private boolean isEmptyString(String name) {
        return name.length() == 0;
    }
}
```

```
private boolean firstIsChar(String name) {
    return isLetter(name.charAt(0));
}

private boolean restIsNameChar(String name) {
    boolean yes = true;
    for (int i = 1; i < name.length(); i++) {
        if (!isNameChar(name.charAt(i))) {
            yes = false;
            break;
        }
    }
    return yes;
}
```

Lösung zu Selbsttestaufgabe 26.2-1:

```
sb.delete(6, 13).insert(20, "un");
```

Lösung zu Selbsttestaufgabe 26.2-2:

```
sb.reverse().delete(0, 2);
```

Lösung zu Selbsttestaufgabe 26.2-3:

Für die Klasse Artikel:

```
public String toString() {
    return "Artikel " + this.liefereName()
        + "Nr: " + this.liefereArtikelnummer()
        + "Preis: " + this.lieferePreis() + "\n";
}
```

Für die Klasse Pflanze:

```
public String toString() {
    return super.toString()
        + "Lagertemperatur: "
        + this.liefereLagertemperatur() + "\n";
}
```