Programmiersprache

Java

2022 / Teil 7

Detlef Wilkening
www.wilkening-online.de

Programmiersprache Java

14 Vererbung 2
14.1 Vererbungs-Hierarchien 2
14.2 Implementation 3
14.3 Schlüsselwort super 4
14.4 Konstruktoren 4
14.5 Überschreiben 5
14.6 Ist-ein Beziehung 7
14.7 Polymorphie 8
14.8 abstract 9
14.9 Casts und instanceof 10
14.10 Klasse „java.lang.Object“ 11
14.11 Anwendung – Beispiel „Obstkorb“ 13
14.12 Interfaces 22
14.13 Modul-Entkopplung 25
14.14 Fazit 27

15 Innere Klassen 28
15.1 Eingebettete static Klassen 28
15.2 Eingebettete nicht static Klassen 29
15.3 Eingebettete Interface’s 31
15.4 Lokale Klassen 31
15.5 Anonyme Klassen 33
15.6 Lambdas 34
15.7 Virtuelle Maschine 37

14 Vererbung

14.1 Vererbungs-Hierarchien

Vererbung bedeutet "*ist ein*".

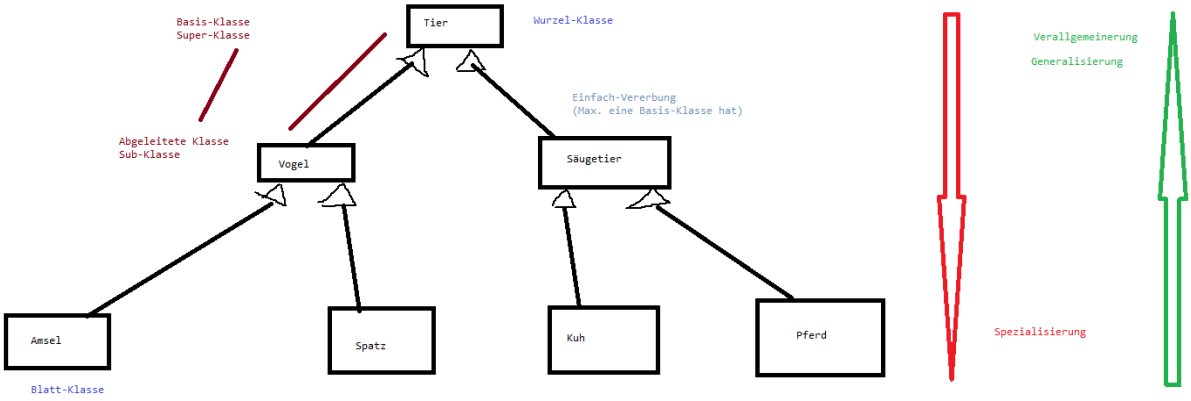
- Ein Hammer ist ein Werkzeug
- Ein Auto ist ein Fahrzeug
- Ein Rechteck ist eine geometrische Figur
- Ein Pferd ist ein Säugetier ist ein Lebewesen
- Ein Spatz ist ein Vogel ist ein Lebewesen
- Eine Amsel ist ein Vogel ist ein Lebewesen

Allgemein:

- Ein B bzw. ein C ist ein A

Hierbei ist:

- A Basisklasse (von B und C)
- B ist abgeleitet von A => B ist ein A => alles was für A gilt, gilt auch für B
- C ist abgeleitet von A => C ist ein A => alles was für A gilt, gilt auch für C



In Richtung der abgeleiteten Klassen findet eine Spezialisierung statt:

- B ist eine Spezialisierung von A
- Ein Pferd ist eine Spezialisierung eines Säugetiers
- Alles, was für Säugetiere gilt (z. B. geboren werden, schwanger sein, sterben), gilt auch für Pferde. All diese Attribute und Funktionen erbt Pferd von Säugetier.

In Richtung der Basis-Klassen findet eine Generalisierung oder Verallgemeinerung statt - Basis-Klassen fassen gemeinsame Dinge der abgeleiteten Klassen zusammen:

- A enthält alle Gemeinsamkeiten von B und C
- Vogel enthält alles Vogel-typische, unabhängig, ob es sich um eine Amsel oder einen Spatz handelt.

Hinweise

- Abgeleitete Klassen (z. B. B und C) sind unabhängig voneinander.
- Von einer Klasse können beliebig viele andere Klassen abgeleitet werden.
- Eine Klasse hat immer nur eine Basisklasse (Einfachvererbung).
  - In Java genau genommen hat eine Klasse immer genau eine Basisklassen

Achtung

Unterscheiden Sie zwischen *„hat ein"* bzw. *„ist implementiert mit"* und *„ist ein"*.

- Eine Person hat einen Namen, ist aber kein Name => Aggregation, Komposition, ...
- Eine Person ist ein Lebewesen, **immer ohne wenn und aber** => Vererbung

14.2 Implementation

Wie wird Vererbung in Java implementiert?

Syntax
[Modifier] class klassenname extends Basisklasse { Klassen-Definition }

```
public class A {

  public void afct() {
    System.out.println("afct in A");
  }

}
```

```
public class B extends A {

  public void bfct() {
    System.out.println("bfct in B");
  }

}
```

```
// Normales A Verhalten
A a = new A();
a.afct();

// Normales B Verhalten
B b = new B();
b.bfct();

// Aber B ist auch ein A, darum 'kann' es alles, was A 'kann'
b.afct();                // Ausgabe: afct in A
```

An diesem Beispiel sehen Sie, dass die Klasse B alle Funktionen von A erbt, d.h. sie ohne weitere Schreibarbeit zur Verfügung stehen. Jede Änderung in A wirkt sich damit sofort auch auf B (und natürlich alle weiteren abgeleiteten Klassen) aus.

Und Sie sehen, dass die Klasse B das **Verhalten** (Methoden) von A erbt, d.h. sie ohne weitere Schreibarbeit zur Verfügung stehen. Jede Änderung in A wirkt sich damit sofort auch auf B (und natürlich alle weiteren abgeleiteten Klassen) aus.

14.3 Schlüsselwort super

In jeder Element-Funktion steht das Schlüsselwort super zur Verfügung, das immer für den Objektanteil der Basisklasse des aktuellen Objektes steht. Beispiel siehe z. B. Konstruktoren.

14.4 Konstruktoren

Konstruktoren werden in Java **nicht** vererbt. Dies macht in den meisten Fällen auch keinen Sinn, da ein Konstruktor immer das erzeugte Objekt initialisieren soll – ein geerbter Konstruktor aber nur den Objektanteil der Basisklasse initialisieren kann. Sie müssen daher für jede Klasse wieder neu Konstruktoren erstellen.

Die Konstruktoren der abgeleiteten und der Basis-Klasse sind automatisch miteinander verkettet, d.h. jeder Konstruktor einer abgeleiteten Klasse ruft als erstes Defaultmäßig den

Standard-Konstruktor der Basisklasse auf.

```
public class A {

  public A() {
    System.out.println("Konstruktor A");
  }

}
```

```
public class B extends A {

  public B() {
    System.out.println("Konstruktor B");
  }

}
```

```
B b = new B();
```

```
Ausgabe
Konstruktor A
Konstruktor B
```

Sie sehen, dass als erstes automatisch der Standard-Konstruktor der Basisklasse aufgerufen wird. Wollen sie, dass ein anderer Konstruktor für die Basisklasse benutzt wird, können sie diesen am Anfang des Konstruktors der abgeleiteten Klasse mit super angeben.

```
public class A {

  public A(int i) {
    System.out.println("Konstruktor A mit " + i);
  }

}
```

```
public class B extends A {

  public B() {
    super(11);
    System.out.println("Konstruktor B");
  }

}
```

```
Ausgabe
Konstruktor A mit 11
Konstruktor B
```

Wenn die Basisklasse keinen Standard-Konstruktor hat, so **müssen** sie in den abgeleiteten Konstruktoren mit super einen Konstruktor angeben.

Sie können in einem Konstruktor mit „this“ auch einen anderen Konstruktor der Klasse anspringen, der dann einen Konstruktor der Basisklasse aufruft.

14.5 Überschreiben

Funktionen von Klassen können in abgeleiteten Klassen überschrieben werden, d.h. sie können für eine abgeleitete Klasse neu definiert werden (gleicher Name, gleiche Parameterliste).

```
public class A {

  public void fct() {
    System.out.println("fct in A");
  }

}
```

```
public class B extends A {

  @Override
  public void fct() {
    System.out.println("fct in B");
  }

}
```

```
A a = new A();
a.fct();                // fct in A
B b = new B();
b.fct();                // fct in B
```

Anmerkung: Bitte ignorieren Sie bitte erstmal die Annotation “@Override” in der Klasse “B”. Wir werden sie gleich besprechen.

Auf die Art und Weise kann eine nicht passende Implementierung einer Basisklasse in einer abgeleiteten Klasse neu implementiert, d. h. überschrieben werden. Ein Beispiel wäre die Methode berechneGehalt in einer Klasse Angestellter und in der abgeleiteten Klasse Vertreter. Ein Vertreter ist sicherlich auch ein Angestellter, d.h. für ihn gelten die Methode getName, getPersonalNo(),..., aber das Gehalt wird bei Vertretern oft anders berechnet.

Oft ist es so, dass die Basisklassen Implementierung gar nicht so schlecht ist, aber eben nicht 100 % passt. Vielleicht bekommt der Vertreter zusätzlich zu einem Festgehalt nur noch einen variablen Anteil hinzu – in diesem Fall wäre die Basisklassen Implementierung ja nicht falsch, sondern eben nur ein Teil der korrekten Implementierung. Darum ist es oft sinnvoll in einer Neu-Implementierung auf die Basisklassen Implementierung zurückzugreifen. Hierbei gibt es zwei ‚*reine*‘ Formen (Korrektur und Filterung), aber natürlich auch beliebige Mischformen.

14.5.1 Korrektur

Falls das Ergebnis der Basisklassen-Elementfunktion nicht 100% passend ist, kann es in einer abgeleiteten Klasse korrigiert werden – denken sie an das Vertreter Beispiel von oben.

Prinzip

```
void f() {
  super.f();          // expliziter Aufruf der Original-Elementfunktion
  // Korrektur        // Korrektur des Ergebnisses
}
```

14.5.2 Filterung

Falls die Basisklassen-Elementfunktion nicht alle Fälle (korrekt) behandelt, können diese vorher

abgefangen und behandelt werden.

Prinzip

```
void f() {
  // Filterung            // Filterung mancher Faelle
  super.f();              // expliziter Aufruf der Original-Elementfunktion
}
```

Die Filterung bezieht sich häufig auf die Übergabeparameter.

14.5.3 Annotation „@Override"

Die Annotation "@Override" vor der überschreibenden Funktion ist optional, sollte von Ihnen aber immer genutzt werden. Mit der Annotation sagen Sie dem Compiler, dass diese Funktion eine andere überschreibt. Ändert sich die zu überschreibende Funktion in der Basisklasse (Name, Signatur,…) , ohne dass Sie das mitbekommen, so würden Sie die Funktion nicht mehr überschreiben. Damit würde sich das Verhalten Ihres Programms ändern – ohne dass Sie das merken. Mit der Annotation "@Override" meldet der Compiler eine fehlende Überschreibung als Compile-Fehler. Dies ist in der Praxis sehr hilfreich.

14.6 Ist-ein Beziehung

Eine Konsequenz aus der Semantik 'Vererbung ist eine ist-ein Beziehung' ist, dass einer Referenz-Variablen der Basisklasse auch ein Objekt einer abgeleiteten Klasse zugewiesen werden kann.

```
// Klasse B ist von A abgeleitet
A a1 = new A();
A a2 = new B();
```

Dies ist ganz im Sinne der Semantik. Ist-ein heißt, dass für ein B Objekt alles gilt, was für ein A Objekt gilt - und daher ein B Objekt auch das Interface von A unterstützt.

Bemerkung – falls Sie das Ganze etwas verwundert, machen Sie sich mal von der ganzen Computerei frei, und betrachten das Ganze mit einem *normalen* Beispiel: Wenn Sie z.B. auf einen Stuhl zeigen und sagen „das ist ein Stuhl", dann wird Ihnen wohl niemand widersprechen. Aber auch die Aussage „das ist ein Möbelstück" wäre ohne Frage richtig. Und genau das gleiche passiert hier: Die Referenz-Variable „a2" sagt mit ihrem statischen Typ „A" das sie ein Möbelstück referenziert (*darauf zeigt*), obwohl sie doch in Wirklichkeit einen Stuhl (ein Objekt vom Typ „B") referenziert (*darauf zeigt*). Aber daran ist nichts Falsches und unwahres - sie sagt nur nicht alles. Aber in vielen Kontexten reicht das. Wir sagen zu unserem Besuch auch „Nimm dir einen Stuhl", und lassen offen, ob er sich in einen Sessel, die gute Coach oder den normalen Holzstuhl setzen soll. Warum auch? Im Prinzip würde es sogar reichen zu sagen „Nimm doch bitte Platz".

Hinweis - man unterscheidet daher auch in den sogenannten statischen und den dynamischen Typ.

- Der statische Typ ist der Typ der Referenz-Variablen. Diesen Typ sieht der Compiler, da er ohne wenn und aber zur Compile-Zeit feststeht und eindeutig bekannt ist - er steht ja als Typ an der Definition der Referenz-Variablen. Im Beispiel ist dies der Typ “A“ der Referenz-Variablen „a1“ und „a2“.
- Dem gegenüber ist der dynamische Typ der echte Typ des Objekts, auf das verwiesen wird. Dieser ist zur Compile-Zeit nicht zwingend bekannt, und muss nicht dem statischen Typ entsprechen. Im Beispiel ist der dynamische Typ des von „a1“ referenzierten Objekts „A“, während es bei „a2“ „B“ ist.

14.7 Polymorphie

Überschreiben und ist-ein-Beziehung zusammen ermöglichen ein Feature, das das Schlüsselkonzept aller OO Sprachen ist: Polymorphie.

```
public class A {

  public void fct() {
    System.out.println("fct in A");
  }

}
```

```
public class B extends A {

  @Override
  public void fct() {
    System.out.println("fct in B");
  }

}
```

```
A a1 = new A();
A a2 = new B();
a1.fct();       // fct in A
a2.fct();       // fct in B - obwohl ueber eine A Referenz-Variab. aufgerufen
```

In Java wird der Funktionsaufruf **erst zur Laufzeit** festgelegt, und zwar in Abhängigkeit vom echten Typ des Objekts, und **nicht** in Abhängigkeit vom Typ der Referenz-Variablen. Dieses Sprachfeature wird mit Polymorphie bezeichnet.

Mit Polymorphie ist gemeint, dass eine Funktion vielgestaltig ist, d. h. in Abhängigkeit vom Kontext unterschiedlich (angepasst) reagiert. Genau genommen reagiert natürlich nicht eine Funktion unterschiedlich, sondern es werden unterschiedliche Funktionen aufgerufen, ohne dass sich der Entwickler um die echten Objekt-Typen und deren verschiedene Funktionen-Implementierungen kümmern muss. Dies ermöglicht es ihm, ähnliche Objekte gleich zu behandeln, ohne Details kennen zu müssen (z.B. welche Klassen es gibt, wie sie heißen, wie sie zu behandeln sind, usw...).

Hinweis – im ersten Augenblick sieht Polymorphie nicht nach was Besonderem aus, sondern eher nur nach einem kleinen Sprachgag – aber dies ist falsch. Es ist **das Schlüsselkonzept** der Objektorientierung. Seine wahre Mächtigkeit erkennt man meist erst in praktischen

Einsätzen, von denen in den weiteren Kapiteln ein paar folgen werden.

14.8 abstract

Es gibt Situation, in denen eine Basisklasse keine sinnvolle Default-Implementierung für eine Element-Funktion anbieten kann – ein Beispiel hierfür findet sich u.a. im Beispiel-Kapitel 14.11. In diesem Fall bekommt die entsprechende Element-Funktion den Modifier abstract, was bedeutet, dass diese Element-Funktion in dieser Klasse nur deklariert, aber nicht implementiert wird.

Sobald mindestens eine Element-Funktion in einer Klasse abstract ist (und sei es auch durch Vererbung – siehe Klasse „B“ im Beispiel), muss auch die Klasse den Modifier abstract bekommen. In einer tieferen abgeleiteten Klasse ohne Modifier abstract **muss** diese Element-Funktion jetzt überschrieben und implementiert werden.

```
public abstract class A {

  public abstract void f();

}
```

```
public abstract class B extends A {
}
```

```
public abstract class C extends B {

  public abstract void f();

}
```

```
public class D extends C {

  public void f() {
    System.out.println("Hallo");
  }

}
```

```
A a = new D();
a.f();            // Ausgabe: Hallo
```

Eine abstrakte Klasse kann damit automatisch nicht mehr instanziiert werden – was ja auch keinen Sinn mehr macht, da sie eine Funktion ohne Implementierung anbietet.

```
public abstract class A {

  public abstract void f();

}
```

```
A a = new A();    // Fehler - A laesst sich nicht instanziieren, da abstract
a.f();            // Welche Funktion sollte das dann auch sein??
```

Eine Klasse darf auch dann abstrakt sein, wenn sie keine abstrakten Funktionen hat.

Typischerweise tritt dieser Fall bei Klassen auf, die keine konkreten Objekte beschreiben, sondern nur allgemeine Beschreibungen für die Gemeinsamkeiten einer *„Objekt-Familie“* sind.

14.9 Casts und instanceof

Mit Casts kann man statische Typen in der Vererbungs-Hierarchie verschieben. Dazu wird der gewünschte Ziel Typ in Klammern vor den Quellausdruck geschrieben. Achtung – dies geht nur, solange die referenzierten Objekte wirklich solche sind. Ansonsten wird eine Exception geworfen.

Mit dem Operator „instanceof“ kann abgefragt werden, ob ein Objekt von einem bestimmten Typ ist.

```
public class A {
}
```

```
public class B extends A {

  public void f() {
    System.out.println("B.f");
  }

}
```

```
A var = new A();
if (var instanceof B) {
  System.out.println("Variable var referenziert ein B Objekt");
  B b = (B)var;
  b.f();
} else {
  System.out.println("Variable var referenziert KEIN B Objekt");
}

var = new B();

if (var instanceof B) {
  System.out.println("Variable var referenziert ein B Objekt");
  B b = (B)var;
  b.f();
} else {
  System.out.println("Variable var referenziert KEIN B Objekt");
}
```

Ausgabe

```
Variable var referenziert KEIN B Objekt
Variable var referenziert ein B Objekt
B.f
```

```
try {
  A a = new A();
  B b = (B)a;
  b.f();
} catch (Exception x) {
  System.out.println(x);
}
```

Ausgabe

```
java.lang.ClassCastException
```

Achtung – Casts innerhalb einer Vererbungs-Hierarchie sind in einer Sprache mit häufig untypisierten Schnittstellen relativ normal. Trotzdem sollte ihnen klar sein, dass Casts kein guter

Programmierstil sind, und auf das Notwendigste beschränkt sein sollten.

Noch extremer ist dies mit der Verwendung von „instanceof“. Im Normallfall sollte die Kenntnis der konkreten Typen hinter einem Basis-Klasse oder einem Interface unnötig sein, da dies z.B. die Erweiterbarkeit und Wiederverwendbarkeit stark einschränkt. Daher sollte die Benutzung von „instanceof“ der gut begründete Ausnahmefall bleiben.

Hinweise:
- Seit Java 17 gibt es als Preview Pattern-Matching für „instanceof“ als Switch-Anweisung, was in den Fällen, wo „instanceof“ dann doch benutzt werden muss, den Code oft viel einfacher macht. Pattern-Matching für „instanceof“ wurde in Kapitel 7.3.2 detailliert besprochen.
- Wird „instanceof“ mit „null“ bzw. einer Null-Variablen aufgerufen, so ist das Ergebnis immer „false“.

14.10 Klasse „java.lang.Object“

Alle Klassen in Java sind **immer** direkt oder indirekt von der Klasse „Object“ abgeleitet. Wenn sie keine Basisklasse angeben, wird automatisch „Object“ als Basisklasse angesetzt.

Die Klasse „Object“ beinhaltet allgemeine Methoden, die für jede Klasse sinnvoll sind, z.B.

clone()	Erzeugt eine flache Kopie des Objekts. Dazu muss das Objekt das Interface Cloneable implementieren und diese Funktion überschreiben. Arrays sind immer kopierbar.
equals(Object)	Vergleicht zwei Objekte auf Identität, d.h. Referenzgleichheit. Für Objektvergleiche, d.h. tiefe Vergleiche bzw. ein spezielles Vergleichsverhalten muss diese Funktion überschrieben werden.
finalize()	Die normale finalize Methode
getClass()	Gibt die ‘Meta-Klasse’ zum Objekt zurück.
hashCode()	Gibt einen Hash-Wert für das Objekt zurück.
toString()	Gibt das Objekt in einer Text-Repräsentation zurück. Diese Funktion wird automatisch z.B. bei Ausgaben auf die Console oder bei Wandlungen in einen String aufgerufen. Siehe auch Kapitel 14.10.1.

Wenn Sie die Funktionen für ihre Klassen anpassen wollen bzw. müssen, d.h. die geerbte Funktionalität nicht ausreicht, müssen Sie sie überschreiben.

Hinweis – die Funktionen „equals“ und „hashCode“ sind nicht ganz unabhängig voneinander. wenn sie die Equals-Funktion überschreiben, **müssen** sie auch die HashCode-Funktion entsprechend überschreiben. Näheres hierzu finden sie z.B. in der offiziellen Java-Doku oder vielen Büchern. Aus Zeitmangel wird dies in der Vorlesung nicht besprochen.

Achtung – das automatische Erben von „Object“ gilt **nicht** für Interfaces (Kapitel 14.12), sondern **nur** für Klassen. Da „Object“ eine Klasse ist, können Interfaces nicht von ihr erben, d.h.

Klassen haben immer genau eine absolute Basisklasse - das ist „Object“. Interfaces sind da anders.

14.10.1 Element-Funktion „Object.toString“

Ich möchte hier noch mal besonders auf die Funktion „toString“ hinweisen. Sie wird immer dann automatisch aufgerufen, wenn eine Wandlung von einem Objekt in einen String notwendig ist. Dies geschieht:

- Bei der Ausgabe eines Objekts auf der Console mit „System.out.print“ bzw. „System.out.println“.
- Bei der Verkettung eines Strings mit einem Objekt mit dem Plus-Operator.

Da die Klasse „java.lang.Object“ eine Default-Implementierung anbietet, kann jedes Objekt immer ausgegeben bzw. in einen String gewandelt werden. Achtung – die Default-Implementierung von Object liefert keinen besonders sinnvolle Text-Repräsentation, wie auch?

```
public class A {
}
```

```
public class Appl {

  public static void main(String[] args) {
    A a = new A();
    System.out.println(a);

    String s = "String: " + a;
    System.out.println(s);
  }
}
```

```
mögliche Ausgabe
A@119c082
String: A@119c082
```

Mit dem Überschreiben der Funktion „toString“ legen sie das Ergebnis der Wandlung fest.

```
public class A {

  public String toString() {
    return "Ich bin ein A-Objekt";
  }

}
```

```
public class Appl {

  public static void main(String[] args) {
    A a = new A();
    System.out.println(a);

    String s = "String: " + a;
    System.out.println(s);
  }
}
```

```
Ausgabe
Ich bin ein A-Objekt
String: Ich bin ein A-Objekt
```

14.10.2 Object als Basistyp

Jede Klasse ist in Java direkt oder indirekt von „java.lang.Object“ abgeleitet. Daher kann jedes Objekt in Java immer einer Referenz-Variablen vom statischen Typ „Object“ zugewiesen werden. Beispiele:

```
Object o1 = new java.util.TreeMap();
Object o2 = new StringBuffer();
Object o3 = new javax.swing.JFrame();
```

Von daher ist „Object“ der kleinste gemeinsame Nenner aller Objekte – wir lassen die elementaren Datentypen mal außen vor. Und von daher werden in Java viele Funktionen auf „Object“ typisiert, zum Beispiel die Funktionen der Container-Klassen, wenn diese nicht typisiert sind. Daher werden in der Praxis häufig Up-Casts in der Klassen-Hierarchie (siehe Kapitel 14.9) benötigt.

14.11 Anwendung – Beispiel „Obstkorb“

14.11.1 Aufgabe

Nehmen wir an, Sie wollen einen Obstkorb implementieren:

- Ein Obstkorb soll einfach mehrere Früchte verschiedener Obstsorten aufnehmen können.
- Jede Frucht hat einen Namen.
- Auch der Obstkorb hat einen Namen.
- Außerdem soll der Obstkorb einen Konsolen Ausgabe folgender Form haben:
  - Name vom Obstkorb
  - Anzahl der Früchte im Obstkorb
  - Darstellung alle Früchte – alphabetisch sortiert nach dem Namen der Frucht
- Die Darstellung einer Frucht besteht aus einem Namen und der Obstsorte.
- Für den Anfang begnügen wir uns mit den zwei Obstsorten „Apfel“ und „Birne“.

Hier eine mögliche Beispiel-Ausgabe eines Obstkorbs mit 5 Früchten:

```
Gewünschte Ausgabe – wenn denn der Obstkorb fertig wäre...
Ich bin der Obstkorb "Geschenk" und enthalte 5 Fruechte:
- Bauchiger Adler (Birne)
- Dickes Schwein (Apfel)
- Fetter Kohl (Birne)
- Gruener Baum (Apfel)
- Saftiger Schmatz (Apfel)
```

Vorgehen – um zu sehen, wie uns Vererbung, „ist-ein“-Beziehung und Polymorphie hier helfen, werden wir das Programm erstmal ohne diese Sprachmittel implementieren, und dann Stück für Stück Sprachmittel für Sprachmittel nutzen, und dann hoffentlich sehen, wie sie uns das Programmierer-Leben erleichtern.

Bemerkung – wem ein Obstkorb mit Früchten zu abstrakt oder zu gesund ist, der möge sich stattdessen eine Angestellten-Verwaltungs-Software für Arbeiter und Vertriebler vorstellen, oder eine Flughafen-Dispositions-Verwaltung für Flugsteige und Tankwagen, oder...

14.11.2 Lösung 1 – ohne Vererbung und ohne Polymorphie

Zuerst brauchen wir Klassen für Äpfel und Birnen, die den Namen halten und sich selbst entsprechend der Aufgabenstellung darstellen können.

```
public class Apple {

  private String name;

  public Apple(String n) {
    name = n;
  }

  public void print() {
    System.out.println("- " + name + " (Apfel)");
  }

  public String getName() {
    return name;
  }

}
```

```
public class Pear {

  private String name;

  public Pear(String n) {
    name = n;
  }

  public void print() {
    System.out.println("- " + name + " (Birne)");
  }

  public String getName() {
    return name;
  }

}
```

Bevor wir zum Obstkorb kommen – dem eigentlichen Knackpunkt des Programms – implementieren wir die „main“ Funktion – und bekommen damit implizit die Schnittstellen-Definition vom Obstkorb.

```
public class Appl {

  public static void main(String[] args) {
    FruitBasket fb = new FruitBasket("Geschenk");
    fb.insert(new Apple("Dickes Schwein"));
    fb.insert(new Pear("Fetter Kohl"));
    fb.insert(new Apple("Saftiger Schmatz"));
    fb.insert(new Apple("Gruener Baum"));
    fb.insert(new Pear("Bauchiger Adler"));
    fb.print();
  }

}
```

Dann brauchen wir den Obstkorb selber, und das wird schwieriger. Da aber die Schnittstelle aus dem „main“ automatisch heraus fällt, fangen wir damit an:

```
public class FruitBasket {

  private String name;

  public FruitBasket(String n) {
    name = n;
  }

  public void insert(Apple apple) {
    ...
  }

  public void insert(Pear pear) {
    ...
  }

  public void print() {
    ...
  }

}
```

Jetzt stellt sich die Frage, wie wir Äpfel und Birnen im Obstkorb speichern können – am besten schon alphabetisch sortiert. Für die dynamische Speicherung von Objekten haben wir in Kapitel todo mehrere Container kennengelernt. U.a. gab es dabei auch eine Klasse „TreeMap“, die über einen Schlüssel (hier bei uns der Name) alphabetisch sortiert.

Hinweis – in der Praxis würde man hierfür natürlich niemals einen assoziativen Container (d.h. einen mit Schlüssel/Wert Paaren) nehmen, sondern einen Container, der die Sortierung automatisch auf den Objekten selber vornimmt. Den gibt es in Java natürlich auch, z.B. mit der Klasse „TreeSet“ in „java.util“, aber a) kennen wir ihn nicht, und b) müssten wir für seinen Gebrauch wissen, wie man unsere Äpfel und Birnen sortierbar bekommt, d.h. wir müssten mit dem Interfaces „Comparable“ arbeiten oder einen eigenen Comparator implementieren. Für beides ist es noch etwas früh – von daher nehmen wir hier die schlechtere Lösung mit der Klasse „TreeMap“.

Da wir aber noch ohne Vererbung und „ist-ein“ Beziehung arbeiten, können wir Äpfel und Birnen nicht in einer TreeMap speichern. Also geben wir der Obstkorb-Klasse für jeden Obsttyp eine eigene TreeMap.

```
import java.util.TreeMap;

public class FruitBasket {

  private String name;

  private TreeMap<Apple> apples = new TreeMap<>();
  private TreeMap<Pear> pears = new TreeMap<>();

  public FruitBasket(String n) {
    name = n;
  }

  public void insert(Apple apple) {
```

```
    apples.put(apple.getName(), apple);
  }

  public void insert(Pear pear) {
    pears.put(pear.getName(), pear);
  }

  public void print() {
    int count = apples.size();
    count += pears.size();
    System.out.println("Ich bin der Obstkorb \"" + name + "\" und enthalte "
                       + count + " Fruechte:");
    ...
  }

}
```

Als Problem bleibt jetzt nur noch die Ausgabe der Früchte – damit sie über alle Früchte alphabetisch ist, müssen beide TreeMaps parallel durchlaufen werden und die jeweils kleinere Frucht (bezogen auf den Namen) muss ausgegeben werden. Das klingt kompliziert – gerade, wenn man an später mit noch mehr Obstsorten und noch mehr TreeMaps denkt – daher sehe ich hier von einer Lösung ab und überlasse diese dem Studenten ;-)

14.11.3 Lösung 2 – immer noch ohne Vererbung und ohne Polymorphie

Wenn Lösung 1 bei der Ausgabe zu kompliziert ist, man aber keine Vererbung und Polymorphie zur Verfügung hat – was macht man dann? Nun, wenn man in Java ein Problem hat, dann macht man eine Klasse daraus.

Das Problem ist hier, dass wir Äpfel und Birnen gleichzeitig verwalten wollen, dass aber noch nicht können bzw. hier mehr wollen. Also schreiben wir eine Klasse, die das für uns macht – und da sie Früchte verwaltet, nennen wir sie „Fruit“.

```
public class Fruit {

  private Apple apple = null;
  private Pear pear = null;

  public Fruit(Apple a) {
    apple = a;
  }

  public Fruit(Pear p) {
    pear = p;
  }

  public String getName() {
    if (apple!=null) {
      return apple.getName();
    }
    return pear.getName();
  }

  public void print() {
    if (apple!=null) {
      apple.print();
      return;
    }
    pear.print();
```

```
  }

}
```

Die Klassen „Apple“, „Pear“ und „Appl“ sind von dieser Änderung nicht betroffen, nur der Obstkorb selber, da er jetzt intern diese Hilfsklasse „Fruit“ benutzt.

```
import java.util.TreeMap;
import java.util.Iterator;

public class FruitBasket {

  private String name;
  private TreeMap<Fruit> fruits = new TreeMap<>();

  public FruitBasket(String n) {
    name = n;
  }

  public void insert(Apple apple) {
    fruits.put(apple.getName(), new Fruit(apple));
  }

  public void insert(Pear pear) {
    fruits.put(pear.getName(), new Fruit(pear));
  }

  public void print() {
    int count = fruits.size();
    System.out.println("Ich bin der Obstkorb \"" + name + "\" und enthalte "
                       + count + " Fruechte:");
    for (Fruit fruit : fruits.values) {
      fruit.print();
    }
  }

}
```

Bevor Sie weitergehen zu Lösung 3, schauen Sie sich Lösung 2 ruhig mal in Ruhe an. Wir haben hier einen wichtigen Grundsatz von Java kennen gelernt, und sehen ein großes Problem bisheriger Programmierung.

Der Grundsatz ist einfach: **„Haben Sie ein Problem, machen Sie eine Klasse draus“**. Im Prinzip ist dies der alte Grundsatz der Software-Entwicklung „Es gibt kein Problem, das sich nicht durch eine weitere Indirektion kleiner machen lässt“ in neuen Gewändern, nämlich dem OO-Kleid.

Ein großes Problem unseres kleinen Programms ist die **Wartbarkeit**: immer wenn wir hier eine neue Obstsorte einführen, müssen wir neben einer weiteren Klasse in unserem kleinen Programm schon mehrere Code-Stellen mit ändern:

- Die Klasse „FruitBasket“ braucht eine weitere „insert“ Funktion.
- Und die Klasse „Fruit“ muss quasi überall angepasst werden.

Und wir haben ja nur ein kleines Programm. Und dass es so relativ wenig lokalisierte Stellen sind, liegt eigentlich schon daran, dass wir die fast gesamte Logik für die Obstsorten in der Klasse „Fruit“ gesammelt haben.

In einem wirklich großen ernsthaften Programm würde bei einer klassischen Programmierung – wie wir sie hier haben – das Ändern (Einfügen, Löschen, Modifizieren) z.B. einer Flughafen-Ressource oft an tausenden von Stellen Code-Änderungen nach sich ziehen. Ein horrender Aufwand, bei dem man sich nie wirklich ganz sicher sein kann, alle relevanten Stellen berücksichtigt zu haben.

14.11.4 Lösung 3 – mit Vererbung, aber noch ohne Polymorphie

Im Prinzip hat sich eine bessere Lösung schon die ganze Zeit aufgedrängt, aber wir haben sie bewußt nicht eingesetzt. Mit Vererbung und den Konsequenzen über „ist-ein“ Beziehung sind einige Dinge viel einfacher – wir können jetzt an vielen Stellen Äpfel und Birnen gleich behandeln.

Die Klasse „Fruit“ bekommt hier einen ganz anderen Zweck: statt fast alle Probleme zu kapseln mutiert sie zu einer sehr einfachen Basisklasse, die – da sie nicht instanziierbar sein soll – abstrakt ist:

```
public abstract class Fruit {
}
```

Am Beispiel des Apfels schauen wir uns den kleinen Unterschied in der Klassen-Definition der Obstsorten an – er besteht nur aus der Angabe der Basisklasse und dem Schlüsselwort „extends“.

```
public class Apple extends Fruit {

  private String name;

  public Apple(String n) {
    name = n;
  }

  public void print() {
    System.out.println("- " + name + " (Apfel)");
  }

  public String getName() {
    return name;
  }

}
```

Und wie sieht der Obstkorb jetzt aus?

```
import java.util.TreeMap;
import java.util.Iterator;

public class FruitBasket {

  private String name;
  private TreeMap<Fruit> fruits = new TreeMap<>();

  public FruitBasket(String n) {
    name = n;
  }

  public void insert(Apple apple) {
```

```
    fruits.put(apple.getName(), apple);
  }

  public void insert(Pear pear) {
    fruits.put(pear.getName(), pear);
  }

  public void print() {
    int count = fruits.size();
    System.out.println("Ich bin der Obstkorb \"" + name
                       + "\" und enthalte " + count + " Fruechte:");

    for (Fruit fruit : fruits.values) {
      if (fruit instanceof Apple) {
        Apple a = (Apple) fruit;
        a.print();
        continue;
      }
      if (fruit instanceof Pear) {
        Pear p = (Pear) fruit;
        p.print();
        continue;
      }
    }
  }

}
```

Unsere TreeMap kann jetzt direkt alle Früchte ohne Wrapper-Klasse aufnehmen – siehe Element-Funktionen „insert“. Und die Fall-Unterscheidung zwischen Äpfeln und Birnen findet sich nur noch in der Schleife der Obstkorb-Ausgabe.

14.11.5 Lösung 4 – mit Vererbung, aber immer noch ohne Polymorphie

Als wir Vererbung eingeführt haben, haben wir noch nicht über Polymorphie gesprochen, sondern statt dessen den Vorteil herausgestellt, dass gemeinsame Funktionen in eine Basisklasse gelegt werden, und abgeleitete Klassen diese einfach erben. Diesen Vorteil können wir hier auch nutzen, in dem wir das Namens-Handling von Äpfeln und Birnen in die Basisklasse „Fruit“ verschieben – hier am Beispiel von „Apple“ verdeutlicht.

```
public abstract class Fruit {

  private String name;

  public Fruit(String n) {
    name = n;
  }

  public String getName() {
    return name;
  }

}
```

```
public class Apple extends Fruit {

  public Apple(String n) {
    super(n);
  }
```

```
  public void print() {
    System.out.println("- " + getName() + " (Apfel)");
  }

}
```

Dies hat den Nebeneffekt, dass sich im Obstkorb die beiden „insert“ Element-Funktionen zu einer zusammenfassen lassen:

```
public void insert(Fruit fruit) {
  fruits.put(fruit.getName(), fruit);
}
```

14.11.6 Lösung 5 – mit Vererbung und mit Polymorphie

Als unschöne Stelle im Programm bleibt die Ausgabe-Funktion des Obstkorbs über.

```
public void print() {
  int count = fruits.size();
  System.out.println("Ich bin der Obstkorb \"" + name
                     + "\" und enthalte " + count + " Fruechte:");

    for (Fruit fruit : fruits.values) {
      if (fruit instanceof Apple) {
        Apple a = (Apple) fruit;
        a.print();
        continue;
      }
      if (fruit instanceof Pear) {
        Pear p = (Pear) fruit;
        p.print();
        continue;
      }
   }
}
```

Sie sieht kompliziert und fehlerträchtig aus, und sie ist wartungsintensiv. Das Problem hier ist, dass wir Objekte unterschiedlicher Klassen in die Hand bekommen und wir jeweils die entsprechende Element-Funktion der jeweiligen Klasse benutzen wollen. Aber genau das liefert uns doch die Polymorphie!

Wir müssen in der Klasse „Fruit“ nur die entsprechende „print“ Funktion zur Verfügung stellen, und sie dann in den abgeleiteten Klassen überschreiben. Und da wir für „print“ in „Fruit“ keine sinnvolle Default-Implementierung kennen, machen wir die Funktion „abstract“.

```
public abstract class Fruit {

  private String name;

  public Fruit(String n) {
    name = n;
  }

  public String getName() {
    return name;
  }

  public abstract void print();

}
```

Für die Obst-Klassen „Apple“ und „Pear“ ändert sich eigentlich gar nichts – aber Sie sollten natürlich die Annotation „@Override“ noch hinzufügen. Aber die Ausgabe-Funktion des Obstkorbs ist auf einmal ganz einfach:

```
public void print() {
  int count = fruits.size();
  System.out.println("Ich bin der Obstkorb \"" + name
                     + "\" und enthalte " + count + " Fruechte:");
  for (Fruit fruit : fruits.values) {
    fruit.pring();
  }
}
```

Und sie enthält keine Abhängigkeiten auf die verwendeten Obstsorten mehr – auf einmal ist sie ganz allgemeingültig.

14.11.7 Erweiterte Aufgabe

Lassen sie sich das mal auf der Zunge zergehen. In unserem ganzen Programm gibt es fast keine Abhängigkeiten mehr auf die verwendeten Obstsorten. Was heißt das für uns, wenn wir z.B. eine neue Sorte „Bananen“ einführen wollen?

Zuerst brauchen wir in Anlehnung an „Apple“ und „Pear“ eine weitere Klasse „Banana“. Die neue Klasse ist vollkommen unabhängig vom restlichen Programm, und wir können sie einfach hinzufügen:

```
public class Banana extends Fruit {

  public Banana(String n) {
    super(n);
  }

  @Override
  public void print() {
    System.out.println("- " + getName() + " (Banane)");
  }

}
```

Und in „main“ benutzen wir sie einfach:

```
public static void main(String[] args) {
  FruitBasket fb = new FruitBasket("Geschenk");
  fb.insert(new Apple("Dickes Schwein"));
  fb.insert(new Pear("Fetter Kohl"));
  fb.insert(new Apple("Saftiger Schmatz"));
  fb.insert(new Apple("Gruener Baum"));
  fb.insert(new Pear("Bauchiger Adler"));
  fb.insert(new Banana("Krumme Wurst"));
  fb.insert(new Banana("Langer Lulatsch"));
  fb.print();
}
```

```
Ausgabe
Ich bin der Obstkorb "Geschenk" und enthalte 7 Fruechte:
- Bauchiger Adler (Birne)
- Dickes Schwein (Apfel)
```

```
- Fetter Kohl (Birne)
- Gruener Baum (Apfel)
- Krumme Wurst (Banane)
- Langer Lulatsch (Banane)
- Saftiger Schmatz (Apfel)
```

Und ansonsten müssen wir nichts machen – das Programm funktioniert einfach!

Unser eigentliches Programm – hier die Klasse „Obstkorb“ – arbeitet nur auf der Basisklasse, und muss – dank Polymorphie – die konkreten Klassen nicht kennen, und ist daher vollkommen unabhängig. Das Programm lässt sich so sehr leicht verändern oder erweitern.

14.12 Interfaces

Interfaces sind spezielle Java-Klassen:

- Sie sind quasi eine auf die Spitze getriebene abstrakte Klasse.
- Sie werden mit dem Schlüsselwort interface deklariert.
- Sie können nur abstrakte Funktionen und Klassen-Variablen enthalten,
  - d.h. sie enthalten weder Implementationen noch Attribute. Häufig enthalten sie nur Klassen-Konstanten.
  - Seit Java 8 dürfen sie sogenannte Default-Implementierungen enthalten – siehe Kapitel 14.12.1.
- Sie haben keine implizite Basisklasse.
- Sie müssen keine Funktionen enthalten, d.h. sie dürfen auch leer sein.
- Von Interfaces werden Klassen mit dem Schlüsselwort „implements“ abgeleitet.
- Eine Klasse kann beliebig viele Interfaces implementieren.
- Es können Referenz-Variablen vom Typ des Interfaces definiert werden.
- Interfaces benötigen – als spezielle Klassen – ihre eigene Quelltext-Datei, die natürlich so heißen muss wie das Interface.
- Auch Interfaces können eine Vererbungs-Hierarchie bilden. Man kann Interfaces von Interfaces mit dem Schlüsselwort „extends“ ableiten.
- Auch überschriebene Funktionen von Interfaces sollten die Annotation „@Override“ bekommen – siehe folgendes Beispiel:

```
public interface Inter {
  public void fct();
}
```

```
public class Impl implements Inter {

   @Override
   public void fct() {
     System.out.println("In fct() von Impl");
   }

}
```

```
Inter in = new Impl();
in.fct();
```

```
Ausgabe
In fct() von Impl
```

Interfaces sind quasi das Java-Sprachmittel für Mehrfachvererbung, wobei eine Klasse nur über die extends Schiene eine Implementierung erben kann. Eine Klasse kann aber mehrere Interfaces implementieren, und damit mehrere Basis-Konzepte erfüllen. Im Gegensatz zu Mehrfachvererbung ist die Mehrfach-Implementierung von Interfaces einfach und unproblematisch.

```
public interface Inter1 {
  public void fct1();
}
```

```
public interface Inter2 {
  public void fct2();
}
```

```
public class Impl implements Inter1, Inter2 {

   @Override
   public void fct1() {
     System.out.println("In fct1() von Impl");
   }

   @Override
   public void fct2() {
     System.out.println("In fct2() von Impl");
   }

}
```

```
Impl impl = new Impl();
Inter1 in1 =impl;
Inter2 in1 =impl;
in1.fct1();
in2.fct2();
```

```
Ausgabe
In fct1() von Impl
In fct2() von Impl
```

Hinweis – in Java ist es möglich, dass ein Interface komplett leer ist. Mit dem Implementieren eines solchen Interfaces bekommt eine Klasse quasi ein Flag, dass irgendein Verhalten gewünscht ist, okay ist, nicht sein soll, oder was auch immer. Ein Beispiel hierfür ist das Interface „Serializable“, mit dem eine Klasse serialisierbar wird. Man nennt solche Interfaces „Marker-Interface“.

14.12.1 Default-Implementierungen

Seit Java 8 ist es möglich, Funktionen in Interfaces zu implementieren. Wir reden dann von sogenannten Default-Implementierungen. Diese Funktionen müssen mit dem Modifier „default“ ausgezeichnet werden. Damit ist es möglich, eine Default-Implementierung im Interface anzubieten, ohne dass die Funktion in allen abgeleiteten Implementierungs-Klassen überschrieben werden muss.

```
public interface Inter {
   public default void fct() {
      System.out.println("Hallo aus einem Interface");
      System.out.println("- Dank Java 8 Default-Implementierung");
   }
}
```

```
public class Impl implements Inter {
}
```

```
public static void main(String[] args) {
   Inter in = new Impl();
   in.fct();
}
```

Ausgabe

```
Hallo aus einem Interface
- Dank Java 8 Default-Implementierung
```

Natürlich können auch die Default-Implementierungen überschrieben werden.

```
public interface Inter {
   public default void fct() {
      System.out.println("Inter.fct");
   }
}
```

```
public class Impl1 implements Inter {
}
```

```
public class Impl2 implements Inter {
   @Override
   public void fct() {
      System.out.println("Impl2.fct");
   }
}
```

```
public static void main(String[] args) {
   Inter in = new Impl1();
   in.fct();
   in = new Impl2();
   in.fct();
}
```

Ausgabe

```
Inter.fct
Impl2.fct
```

Implementiert eine Klasse mehrere Interfaces mit der gleichen Funktion, die in mindestens zwei Interfaces eine Default-Implementierung hat – so muss die Klasse die Funktion überschreiben, da sonst nicht eindeutig ist, welche Default-Implementierung genommen werden soll. Natürlich kann die überschriebene Funktion dann auf die Default-Implementierung mit „super“ und Angabe des Interface zugreifen – siehe Beispiel.

```
public interface Inter1 {
   public default void fct() {
      System.out.println("Inter1.fct");
```

```
    }
}
```

```
public interface Inter2 {
    public default void fct() {
        System.out.println("Inter2.fct");
    }
}
```

```
// Compile-Error ohne Override fct
public class Impl1 implements Inter1, Inter2 {
}
```

```
public class Impl2 implements Inter1, Inter2 {
    @Override
    public void fct() {
        System.out.println("Impl2.fct");
        Inter1.super.fct();
    }
}
```

```
public static void main(String[] args) {
    Impl2 impl = new Impl2();
    Inter1 in1 = impl;
    in1.fct();
    Inter2 in2 = impl;
    in2.fct();
}
```

Ausgabe

```
Impl2.fct
Inter1.fct
Impl2.fct
Inter1.fct
```

14.12.2 Funktionale Interfaces

Eine weitere Neuigkeit von Java 8 sind die funktionalen Interfaces. Alle Interfaces, die genau eine Funktion enthalten, sind funktionale Interfaces – egal ob die Funktion geerbt oder direkt enthalten ist, egal ob die Funktion eine Default-Implementierung hat oder nicht. Funktionale Interfaces sind für das neue Sprachmittel Lambdas (siehe Kapitel 15.6) wichtig. Wir zeichnen diese Interfaces mit der Annotation „@FunctionalInterface“ aus – siehe Beispiel. Die Annotation ist nicht notwendig, sollte aber gemacht werden.

```
@FunctionalInterface
public static interface Inter {
    public void fct();
}
```

Ansonsten wird die Bedeutung der funktionalen Interfaces im Kapitel 15.6 über Lambdas näher erläutert.

14.13 Modul-Entkopplung

Vererbung und Polymorphie sind auch ein Mittel, um Module voneinander zu entkoppeln, d.h.

die Module unabhängig voneinander zu machen. Schauen wir uns hierzu mal ein Beispiel an:

Das Benutzer-Interface unterliegt in der Praxis oft häufigeren Änderungen. Damit Änderungen im Benutzer-Interface nicht Änderungen im gesamten Programm nach sich ziehen, versucht man die eigentliche Programm-Logik und das Benutzer-Interface in Schichten aufzuteilen.

In einer solchen Architektur gibt es eine klare Abhängigkeits-Beziehung: hier kennt das UI die BL, aber umgekehrt nicht! Die BL stellt Dienste zur Verfügung, die vom UI benutzt werden können. Die BL kennt aber kein konkretes UI, und darf daher auch keine Benutzer-Interaktion ausführen!

Was ist aber nun, wenn z.B. die BL während der *Arbeit* Eingaben benötigt, z.B. einen Dateinamen, einen Parameter, ein Passwort oder sonst was? Sie darf ja keine Benutzer-Interaktion ausführen, und kann das UI auch nicht dazu auffordern (da sie es nicht kennt). Was macht man dann?

Formulieren wir das Problem etwas anders, vielleicht fällt dann die Lösung leichter: Die BL-Schicht darf nicht selber Benutzer-Interaktion machen, d.h. muss sie dies indirekt machen, z.B. mittels eines Funktions-Aufrufs. Da sie die-UI Schicht nicht kennt, kann sie dort keine Funktion direkt aufrufen. Sie muss also eine Funktion aufrufen, die in der BL-Schicht bekannt ist, aber in einer unbekannten UI-Schicht ausgeführt wird.

Sehen Sie die Lösung?

- In der BL-Schicht muss eine Funktion bekannt sein, damit die BL-Schicht sie nutzen kann, aber sie kann in der BL-Schicht nicht implementiert sein.
- In der UI-Schicht muss genau diese Funktion dann implementiert sein, und sie muss quasi über die BL-Funktion aufrufbar sein.

Das klingt doch wie Vererbung und Polymorphie:

- In der BL-Schicht wird ein Interface benötigt, dass die Parameter-Hol Funktion definiert.
- In der UI-Schicht muss sich eine UI-Klasse von diesem Interface ableiten und die Parameter-Hol Funktion implementieren.
- Z.B. beim Aufruf der BL-Schicht könnte jetzt das Objekt, das das BL-Interface implementiert, mitgegeben werden.

Und hier das Ganze als Quelltext:

```
package bl;

public interface GetUserParameterInterface {

  public String get();

}
```

```
package bl;

public class BusinessLogic {

  public String doit(GetUserParameterInterface gupi) {
    String parameter = gupi.get();
```

```
    return "\"" + parameter + "\"";
  }

}
```

```
package ui;

import java.io.BufferedReader;
import java.io.InputStreamReader;

import bl.BusinessLogic;
import bl.GetUserParameterInterface;

public class UserInteraction implements GetUserParameterInterface {

  private BusinessLogic bl = new BusinessLogic();

  public void doit() {
    System.out.println("Starte Verarbeitung...");
    String s = bl.doit(this);
    System.out.println("Ergebnis: " + s);
  }

  @Override
  public String get() {
    try {
      System.out.println("Bitte geben Sie einen String ein:");
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader reader = new BufferedReader(isr);
      return reader.readLine();
    } catch (Exception x) {
    }
  return "";
  }

}
```

```
import ui.UserInteraction;

public class Appl {

  public static void main(String[] args) {
    UserInteraction ui = new UserInteraction();
    ui.doit();
  }

}
```

Mögliche Ausgabe

```
Starte Verarbeitung...
Bitte geben Sie einen String ein:
Hallo Welt
Ergebnis: "Hallo Welt"
```

Hinweis – um die Trennung zwischen BL- und UI-Schicht sauber darzustellen, sind die entsprechenden Klassen (bzw. Interfaces) in entsprechende Packages „bl“ und „ui“ eingefügt.

14.14 Fazit

Wenn Sie Vererbung und Polymorphie einsetzen, dann hat das Hinzufügen oder Entfernen oder Verändern einer Klasse fast keine Auswirkungen auf ihr eigentliches Programm – ausgenommen sind die Stellen an denen die Objekte erzeugt werden, aber auch da kann man

sich helfen.

Immer wenn Sie es schaffen ihr Programm, ihre Programm-Struktur oder ihre Programm-Architektur auf Basisklassen (oder Interfaces) aufzubauen, dann haben Sie gewonnen. Sie haben leichtes Spiel mit Veränderungen, ohne dass sie ihren ganzen Code nach Änderungen durchforsten müssen.

Vererbung und Polymorphie sind die Schlüsselkonzepte von OO!

15 Innere Klassen

Seit Java 1.1 können in Java Klassen und Interfaces ineinander verschachtelt werden. Man nennt sie innere Klassen, eingebettete Klassen oder auch verschachtelte Klassen. Es gibt mehrere Sorten von eingebetteten Klassen:

1. **Member-Klassen**

   Eingebettete static Klassen (oder auch „statische Member-Klassen“) sind relativ normale Klassen, d.h. auch Top-Level Klassen, die logisch einer anderen Klasse zugeordnet sind, und auch auf deren private Elemente zugreifen dürfen.

   Eingebettete nicht static Klassen (oder auch „Member-Klassen“) sind sogenannte Element-Klassen, deren Instanzen immer einer umgebenden Klassen-Instanz zugeordnet sind.

   Eingebettete Interfaces (oder auch „Member-Interfaces“) sind vergleichbar zu eingebetteten static Klassen, d.h. sie sind Top-Level-Interfaces, die logisch einer anderen Klasse zugeordnet sind, und auch auf deren private Elemente zugreifen dürfen. Interfaces sind implizit immer static.

2. **Lokale Klassen** sind Klassen die innerhalb eines Code-Blocks definiert sind, und auch nur dort benutzt werden können. Obwohl doch etwas anderes, haben sie viele Eigenschaften mit eingebetteten nicht static Klassen gemeinsam.

3. **Anonyme Klassen** sind lokale Klassen ohne Namen.

Seit Java 8 gibt es in Java „Lambdas“, die eine Art Kurzschreibweise von anonymen Klassen sind – siehe Kapitel 15.6.

15.1 Eingebettete static Klassen

Eingebettete static Klassen:

- Benötigen den Modifier „static“.
- Zugriff über Kombination der Klassen-Namen mit trennendem Punkt.
- Verhalten sich wie normale Klassen (Top-Level-Klassen).

- Können beliebig tief geschachtelt werden.
- Sind logisch der (den) umgebenden Klasse(n) zugeordnet.
- Dürfen auch auf private Elemente der umgebenden Klasse(n) zugreifen.
- Haben direkten Zugriff auf static Elemente der umgebenden Klasse(n).
- Die umgebende Klasse hat auch Zugriff auf private Elemente der inneren Klasse.

```
public class OuterClass {

  private String pstr = "Private aeussere Element-Variable";
  private static String pocv = "Private aeussere Klassen-Variable";

  public OuterClass() {
    System.out.println("Konstruktor OuterClass");
    System.out.println(picv);
    new StaticInnerClass(this);
  }

  public static class StaticInnerClass {
    private static String picv = "Private innere Klassen-Variable";
    public StaticInnerClass(OuterClass o) {
      System.out.println("Konstruktor StaticInnerClass");
      System.out.println(o.pstr);
      System.out.println(pocv);
     }
   }

}
```

```
OuterClass o = new OuterClass();
OuterClass.StaticInnerClass i = new OuterClass.StaticInnerClass(o);
```

Ausgabe

```
Konstruktor OuterClass
Private Innere Klassen-Variable
Konstruktor StaticInnerClass
Private aeussere Element-Variable
Private aeussere Klassen-Variable
Konstruktor StaticInnerClass
Private aeussere Element-Variable
Private aeussere Klassen-Variable
```

15.2 Eingebettete nicht static Klassen

Eingebettete nicht static Klassen:

- Heißen auch Element-Klassen.
- Sind logisch der umgebenden Klasse zugeordnet.
- Ihre Objekte sind immer mit genau einem Objekt der umgebenden Klasse verbunden.
- Dürfen auch auf private Elemente der umgebenden Klasse zugreifen.
- Dürfen keine static Elemente enthalten.
- Dürfen nicht den Namen einer umgebenden Klasse bzw. Packages haben.

```
public class OuterClass {

  private String pstr;

  public OuterClass(String s) {
```

```
    System.out.println("Konstruktor OuterClass");
    pstr = s;
    new InnerClass();
  }

  public class InnerClass {
    public InnerClass() {
      System.out.println("Konstruktor InnerClass");
      System.out.println(pstr);
    }
  }

}
```

```
OuterClass o = new OuterClass("Call in fct");
```

Ausgabe

```
Konstruktor OuterClass
Konstruktor InnerClass
Call in fct
```

Im obigen Beispiel greift der Konstruktor direkt auf eine Objekt-Variable der umgebenden Klasse zu, obwohl hier scheinbar gar kein Objektbezug vorhanden ist. Diesen Objektbezug stellt der Compiler automatisch her. Wird ein Objekt der Elementklasse erzeugt, übergibt der Compiler automatisch die this Referenz des aktuellen Objekts als Objektbezug. Ausserdem erzeugt der Compiler automatisch in der Elementklasse immer ein Attribut für die Referenz auf das zugeordnete umgebenden Klassenobjekt.

Wird versucht, eine Elementklasse in einer anderen Klasse zu erzeugen, so meldet der Compiler einen Fehler, da das aktuelle Objekt nicht den richtigen Klassentyp hat.

```
public class A {
  public void fct() {
    new OuterClass.InnerClass("Call from A");    // Compiler-Error
  }
}
```

Um ein Objekt der Elementklasse außerhalb der umgebenden Klasse erzeugen zu können, muss man dem Compiler den impliziten Parameter explizit angeben. Dafür wurde in Java eine spezielle Syntax für den Operator „new“ eingeführt:

```
OuterClass o = new OuterClass("Call from every-wbere");
o.new InnerClass();
```

Ausgabe

```
Konstruktor OuterClass
Konstruktor InnerClass
Call from every-where
Konstruktor InnerClass
Call from every-where
```

Dieser „new“ Operator wird wie eine Elementfunktion für das zuzuordnende Objekt aufgerufen. Der Klassenname der Elementklasse muss nicht weiter referenziert werden, da automatisch der Namensraum der umgebenden Klasse benutzt wird.

15.3 Eingebettete Interface's

Eingebettete Interface's:

- sind äquivalent zu eingebetteten static Klassen
- sich wie normale Interfaces
- können beliebig tief geschachtelt werden
- sind logisch der (den) umgebenden Klasse(n) zugeordnet
- sind implizit immer static - das Schlüsselwort kann daher weggelassen werden

```
public class OuterClass {

  public static interface InnerInterface {
    public void fct();
  }

}
```

```
public class Concrete implements OuterClass.InnerInterface {

  @Override
  public void fct() {
    System.out.println("In Concrete.fct()");
  }

}
```

```
OuterClass.InnerInterface ii = new Concrete();
ii.fct();
```

Ausgabe

```
In Concrete.fct()
```

15.4 Lokale Klassen

Eine lokale Klasse wird innerhalb eines Code-Blocks definiert, und ist nur innerhalb desselben bekannt. Im Prinzip ist eine lokale Klasse eine spezielle Version einer eingebetteten Klasse, denn jeder Code-Block existiert in Java innerhalb einer Klasse, und so hat auch eine lokale Klasse einen Klassen- bzw. Objektbezug. Daher können die Regeln für eingebettete Klassen erstmal auf lokale Klassen übertragen werden.

- Lokale Klassen sind nur in dem sie definierenden Block bekannt.
- Lokale Klassen können nicht als public, protected, private oder static definiert werden.
- Lokale Klassen dürfen auch auf private Elemente der umgebenden Klasse zugreifen.
- Lokale Klassen dürfen keine static Elemente enthalten.
- Lokale Klassen können auf Variablen, Parameter und Ausnahmen im Sichtbarkeit des definierenden Blocks zugreifen, wenn dieses als „final" definiert sind.
- Lokale Klassen können keine Interfaces sein.
- Lokale Klassen dürfen nicht den Namen einer umgebenden Klasse bzw. Package's haben.

```
public class Normal {

  private String str = "String Attribute in Normal";
```

```
  public void f() {
    final int i = 42;

    class LocalClass {
      public LocalClass() {
        System.out.println("Konstruktor LocalClass");
        System.out.println(str);
        System.out.println(i);
      }
    }

  LocalClass l = new LocalClass();
  }
}
```

```
Normal n = new Normal();
n.f();
```

Ausgabe
```
Konstruktor LocalClass
String Attribute in Normal
42
```

Hinweis - eine lokale ist gegenüber einer eingebetteten Klasse vorzuziehen, wenn die Klasse nur in einer Funktion benötigt wird.

15.4.1 Externe Verwendung

Selbst wenn eine lokale Klasse nur innerhalb des sie definierenden Blocks bekannt ist, so können Objekte der Klasse auch woanders benutzt werden.

```
public interface Interface {
  public void f();
}
```

```
public class Normal {

  public Interface getInterfaceObject {
    class LocalClass implements Interface {
      @Override
      public void f() {
        System.out.println("LocalClass.f()");
      }
    }
    return new LocalClass();
  }

}
```

```
Normal n = new Normal();
Interface i = n.getInterfaceObject();
i.f();
```

Ausgabe
```
LocalClass.f()
```

15.5 Anonyme Klassen

Anonyme Klassen sind lokale Klassen ohne Namen. Sie können direkt an der Stelle, wo ein Objekt von ihnen benötigt und erzeugt wird, ohne Namen definiert werden. Im Gegensatz zu der Definition einer lokalen Klasse ist die Definition einer anonymen Klasse ein Ausdruck, und kann daher Teil eines größeren Ausdrucks (z.B. eines Funktionsaufrufs) sein.

- Anonyme Klassen haben nur den automatischen Default-Konstruktor.
- Objekte anonymen Klassen, die in einer Element-Funktion erzeugt werden, haben automatischen Objekt-Bezug zum aktuellen Objekt der umgebenden Klasse – d.h. dem Objekt, für das die Element-Funktion aufgerufen wurde (dem „this“ der Element-Funktion) – siehe Beispiel.
- Objekte anonymen Klassen, die in einer Klassen-Funktion erzeugt werden, haben keinen Objekt-Bezug.

Syntax

new Basisklasse () { Klassendefinition }

```
public interface Inter {
  public void f();
}
```

```
public class Appl {

  private String s = "Attribute in Appl";

  public static void fct(Inter i) {
    System.out.println("Appl.fct bekommt Objekt vom Typ Inter uebergeben");
    i.f();
  }

  public static void main(String[] args) {
    fct(new Inter() {
      @Override
      public void f() {
        System.out.println("- f() von anonymer Klasse");
        //System.out.println("- - Attribut-Zugriff: \"" + s + "\""); Fehler
        System.out.println("- - kein Attribut-Zugriff");
      }
    });

    Appl appl = new Appl();
    appl.doit();
  }

  public void doit() {
    fct(new Inter() {
      @Override
      public void f() {
        System.out.println("- f() von anonymer Klasse");
        System.out.println("- - Attribut-Zugriff: \"" + s + "\"");
      }
    });
  }

}
```

Ausgabe

```
Appl.fct bekommt Objekt vom Typ Inter uebergeben
```

```
- f() von anonymer Klasse
- - kein Attribut-Zugriff
Appl.fct bekommt Objekt vom Typ Inter uebergeben
- f() von anonymer Klasse
- - Attribut-Zugriff: "Attribute in Appl"
```

Hinweis – anonyme Klassen mögen seltsam und wie ein Spezialfall wirken, aber sie sind z.B. als Implementierung von Listener-Interfaces bzw. -Klassen beim Event-Handling in zum Beispiel Swing das tägliche Brot. Wir werden das bald sehen. Seit Java 8 werden anonyme Klassen weniger verwendet, sondern oft durch Lambdas ersetzt – siehe Kapitel 15.6.

15.6 Lambdas

Im Kapitel 9.4 über das Java 8 Stream-API haben wir Lambdas schon ganz kurz kennengelernt. Auch hier werden wir das Sprachmittel Lambdas nicht tiefer betrachten – das sprengt leider mal wieder den Rahmen der Vorlesung. Stattdessen werden wir hier kurz den technischen Hintergrund der Lambdas betrachten, da sie eine Art von anonymen Klassen sind. Achtung – im Detail wird ein Lambda vom Compiler und der JVM nicht exakt als anonyme Klasse umgesetzt, sondern etwas effizienter. Aber das interessiert in dem hier betrachteten Rahmen nicht.

Denken Sie zurück an die anonymen Klassen. Wenn wir mit einer anonymen Klasse ein Interface implementieren, das nur eine Funktion hat, dann sieht das ungefähr so aus:

```
public interface Inter {
  public void f();
}
```

```
public static void fct(Inter i) {
  i.f();
}
```

```
fct(new Inter() {                        // (*)
  @Override
  public void f() {
    // Tue was
  }
});
```

Um die Funktion „fct“ aufzurufen implementieren, erzeugen wir on-the-fly in (*) eine anonyme „Inter“ implementierende Klasse, erstellen ein Objekt dieser Klasse, und rufen damit die Funktion „fct“ dann auf. Hierbei ist eigentlich der meiste Code von (*) überflüssig.

- Es kann sich ja nur um eine Klasse handeln, die „Inter“ implementiert, denn dieser Typ wird von „fct“ erwartet.
- Natürlich benötigen wir ein Objekt
- Und da das Interface „Inter“ ja nur eine Funktion hat, wollen wir die doch wohl überschreiben. Alles andere macht ja keinen Sinn.

Nur die eigentliche Implementierung in „f“ – hier mit „Tue was“ beschrieben – ist der sinnvolle Code. Alles andere schreiben wir nur für den Compiler.

Und an der Stelle kommen die Lambdas ins Spiel. Sie stellen eine Kurzschreibweise hierfür da, die sich rein auf die Parametername der Funktion und die Implementierung konzentriert. Alles andere fällt weg – damit wird der Code viel kürzer, übersichtlicher und prägnanter.

```
public interface Inter {
   public void fct(int a);
}
```

```
public static void doit(Inter in) {
   in.fct(42);
}
```

```
// Klassische Top-Level Klasse als Implementierung
public class Impl implements Inter {
   @Override
   public void fct(int a) {
      System.out.println("Impl.fct: " + a);
   }
}
```

```
public static void main(String[] args) {

   // Nutzung der klassischen Klasse
   doit(new Impl());

   // Anonyme Klasse
   doit(new Inter() {
      @Override
      public void fct(int a) {
         System.out.println("Anonyme Klasse: " + a);
      }
   });

   // Lambda
   doit(a -> System.out.println("Lambda: " + a));
}
```

```
Ausgabe
Impl.fct: 42
Anonyme Klasse: 42
Lambda: 42
```

Lambdas:

- Eingeleitet wird das Lambda mit der Auflistung der Parameter – im Beispiel ist dies der Parameter „a“. Ist kein Parameter vorhanden, werden hier die leeren Klammern genommen – siehe nächstes Beispiel.
- Dann folgt der Pfeil „->“, der das Lambda identifiziert.
- Hinter dem Lambda folgt die eigentliche Implementierung der überschriebenen Funktion. Definiert die Funktion einen Rückgabe-Typ, so wird das Ergebnis des Implementierungs-Ausdrucks automatisch zurückgegeben – siehe übernächstes Beispiel.

Lambdas funktionieren natürlich nur, wenn das implementierte Interface nur eine Funktion hat. Ansonsten könnte der Compiler nicht deduzieren, welche Funktion überschrieben wird. Da Lambdas den Code sehr vereinfachen und für viele Situation sehr hilfreich sind (z.B. bei der Stream-API), und sie auf Interfaces mit einer Funktion basieren – sind diese Interfaces sehr wichtig. Wir nennen sie darum seit Java 8 funktionale Interfaces (siehe Kapitel 14.12.2) und

zeichnen sie daher auch mit der entsprechenden Annotation aus.

15.6.1 Lambda ohne Parameter

```
@FunctionalInterface
public interface Inter {
   public void fct();
}
```

```
public static void doit(Inter in) {
   in.fct();
}
```

```
public static void main(String[] args) {

   // Anonyme Klasse
   doit(new Inter() {
      @Override
      public void fct() {
         System.out.println("Anonyme Klasse");
      }
   });

   // Lambda
   doit(() -> System.out.println("Lambda"));
}
```

Ausgabe
```
Anonyme Klasse
Lambda
```

15.6.2 Lambda mit zwei Parametern und Rückgabe

```
@FunctionalInterface
public interface MathOperation {
   public int exec(int x, int y);
}
```

```
public static void doit(int a1, int a2, MathOperation op) {
   System.out.println(op.exec(a1, a2));
}
```

```
public static void main(String[] args) {
   doit(20, 22, new MathOperation() {
      @Override
      public int exec(int x, int y) {
         return x+y;
      }
   });

   doit(6, 7, (a,b) -> a*b);
   doit(84, 2, (a,b) -> a/b);
}
```

Ausgabe
```
42
42
42
```

15.6.3 Probleme mit „var“ bei Lambdas

Natürlich funktionieren die Typ-Deduktion und die Nutzung von Lambdas auch mit normalen Variablen, aber natürlich nicht mit der automatischen Typ-Inferenz mit „var“ seit Java 10 (siehe Kapitel 5.3.1). Bei einer normalen Variablen kann der Compiler die Information für die Lambda Generierung aus dem Typ der Variablen herauslesen. „var“ setzt aber ja gerade voraus, dass der Initialwert den Typ bestimmt – hier beißt sich die Schlange dann in den Schwanz.

```
@FunctionalInterface
public interface Inter {
   public void fct(int a);
}
```

```
Inter lbd1 = x -> System.out.println(x);     // OK

var lbd2 = x -> System.out.println(x);       // Compile-Error - zu viel verlangt
```

15.7 Virtuelle Maschine

Die virtuelle Java Maschine kennt keine eingebetteten Klassen. Damit sie trotzdem den erzeugten Byte Code verarbeiten kann, wendet der Compiler einen kleinen Trick an: er erzeugt statt der eingebetteten Klassen scheinbare Top-Level Klassen, deren Namen sich aus dem Namen der umgebenden Klasse, einem $ Zeichen und dem Namen der eingebetteten Klasse ergibt.

```
public class OuterClass {

  public static class StaticInnerClass {
  }

}
```

So finden sich nach einem solchen Code-Stück im entsprechenden Byte-Code Verzeichnis die Dateien:

- OuterClass.class
- OuterClass$StaticInnerClass.class

wieder. Schauen Sie sich das Ausgabe-Verzeichnis ruhig mal an.