

Objektorientierte Modellierung und Programmierung (Teil 2)

Phasen der objektorientierten Software-Entwicklung

Objektorientierte Analyse (OOA)

- Ziel: allgemeines Systemmodell erstellen ohne Implementierungsdetails
- Erfassung der Anforderungen und Beschreibung, was das zu entwickelnde System machen soll (führt zu sog. Pflichtenheft)
- Erstellung eines objektorientierten Analysemodells (OOA-Modell) unter Verwendung von UML:
 - statisches Modell: Beschreibung der Struktur des Systems, insb. Mit Klassendiagrammen (Klassennamen, Attributnamen, Methodennamen, Klassenbeziehungen); Objektkonfigurationen mit Objektdiagrammen
 - dynamisches Modell: Anwendungsfalldiagramm

Objektorientierter Entwurf (OOE), auch: obj.-orient. Design (OOD)

- Erstellung eines objektorientierten Entwurfsmodells (OOE/D-Modell)
 - statisches Modell: Verfeinerung des OOA-Klassendiagramms mit Implementierungsdetails (z. B. Spezifikation von Attributen, Methoden, Klassenbeziehungen, Ergänzung weiterer für die Implementierung erforderlicher Klassen) zum sog. OOE-Klassendiagramm
 - dynamisches Modell: „zeitliches Verhalten“ des Systems, insb. mit Interaktions- u. Zustandsdiagrammen
 - Planung und Beschreibung, wie das geplante System die Anforderungen realisiert

Objektorientierte Programmierung (OOP)

- im engeren Sinne: systematische Überführung der zuvor entwickelten UML-Diagramme in Quelltext der verwendeten objektorientierten Programmiersprache (teilweise softwaregestützt möglich)
- im weiteren Sinne: gesamter Prozess

Motivation für die Verwendung von UML in allen Phasen

- Beschreibung verschiedener Ausbaustufen des geplanten Systems
- Ideen können auf der Konzeptebene diskutiert werden
- Idee wird zum Modell, Modell wird zum Programm
- Industriestandard
- bewährt auch im praktischen Einsatz interdisziplinärer Projektteams

Beispiel: Bibliotheksverwaltung

Vom Anwender zu Anwendungsfällen

1. Schritt: Beschreibung der Nutzer und ihrer Anforderungen

- Analyse aus Anwendersicht:
 - Akteure (engl. actor) identifizieren
 - Akteur: Anwender (Mensch, Maschine, Programm) des zu entwerfenden Systems in einer bestimmten Rolle
 - tritt mit dem zu entwerfenden System in Interaktion
 - hat Anforderungen an dieses System
 - Anwendungsfälle (engl. use cases) identifizieren und beschreiben
 - Anwendungsfall: Aufgabe, die ein Akteur mit Hilfe des zu entwerfenden Systems lösen will / muss
 - Identifikation durch Textanalyse der Aufgabenstellung im Hinblick auf Zeitwortphrasen (z. B. ... leiht Buch aus ...)
 - detaillierte textuelle Beschreibung der Abläufe (sog. Szenarien)

UML-Anwendungsfalldiagramm (engl. use case diagram)

- Modellierungselemente: Akteure als Strichfiguren, Anwendungsfälle als Ovale, Beteiligung eines Akteurs an einem Anwendungsfall als Linie.

2. Schritt: Klassenkandidaten identifizieren

- Ausgangspunkt dafür:
 - ausführliche, textuelle Beschreibung der Aufgabenstellung
 - ausführliche, textuelle Beschreibung der Anwendungsfälle
- Textanalyse („Abbot’s noun approach“)
- Substantive im Text:
 - Kandidaten für Klassen, wenn damit etwas Systemrelevantes beschrieben wird, über das mehrere Informationen zu speichern sind, z. B. Mitglied, Buch
 - Kandidaten für Attribute, wenn damit nur einzelner Wert beschrieben wird, z. B. Name, Mitgliedsnummer
- Verben im Text:
 - Kandidaten für Methoden, wenn damit etwas Systemrelevantes beschrieben wird

Identifizierung von Klassenkandidaten

- Klassen für einzelnes Mitglied und Verwaltung von Mitgliedern
- entsprechend für Mitarbeiter, Buch, Hörbuch, Spiel und Ausleihvorgang
- Mitglied hat Attribute Name, Privatanschrift, Mitgliedsnummer
- Klassen zur Verwaltung der Mitglieder, Medien und Ausleihvorgänge sollen die Ausgabe auf dem Bildschirm ermöglichen, haben also z. B. eine Methode print()
- Daraus folgt, dass jedes Objekt der Klassen Mitglied, Mitarbeiter, Buch, Hörbuch, Spiel, Ausleihvorgang auch einzeln auf dem Bildschirm ausgegeben werden kann, Klassen haben also z. B. Methode print()
- Bibliothek beschreibt Gesamtsystem (z. B. denkbarer Name einer Klasse, die im Falle einer Java-Implementierung die main()-Methode enthält).

Von Klassenkandidaten zum Klassendiagramm: CRC-Karten

3. Schritt: Klassenkarten entwerfen

- CRC-Kartentechnik (CRC – Classes, Responsibilities, Collaborators)
- Didaktisches Hilfsmittel zur Vermittlung objektorientierten Denkens.
- Erstellung je einer Klassenkarte je gefundenem Klassenkandidaten.
- Informelle Erfassung
 - wofür Klasse zuständig ist (links) – das führt später zu Attributen und Methoden,
 - mit Objekten welcher anderen Klassen zusammengearbeitet wird, um das Systemziel zu erreichen – führt zu sog. Klassenbeziehungen,
 - Quellen: Klassenkandidaten, Aufgabenbeschreibung, Anwendungsfallbeschreibungen
- Aufbau:

<Klassenname>	
<Zuständigkeiten>	<Zusammenarbeit mit>
- Zwischenschritt:
 - Prüfen und ggf. Modifizieren der CRC-Karten der Klassenkandidaten.
 - Klassenkandidaten mit mehr als 4 bis 5 Zuständigkeiten sollten aufgeteilt werden.
 - Klassenkandidaten mit 0 bis 1 Zuständigkeiten sollten verworfen werden, sofern sich ein anderer Klassenkandidat findet, der diese Zuständigkeiten sinnvoll übernehmen kann.
- Dann: Anordnung der Karten z. B. an Tafel nach Zusammengehörigkeit, ggf. Verbindungslinien zwischen zusammenarbeitenden Klassen.
- Führt zu einer ersten Version eines Klassendiagramms.
- Dann: Überführung in UML-Klassendiagrammschreibweise.

UML-Klassendiagramm

- beschreibt die im betrachteten System vorkommenden Klassen und deren Beziehungen untereinander
- Bislang: nur einzelne Klassen
 - strukturelle Eigenschaften, innere Struktur: Attribute
 - Verhalten: Methoden
- Ab jetzt auch: Beziehungen zwischen den Objekten von Klassen
- Varianten:
 - Assoziationen, Aggregationen, Kompositionen; können genauer modelliert werden durch
 - Multiplizitäten
 - Rollennamen und Navigationspfeile
 - Spezialisierungen/Generalisierungen

Assoziationen

- (zweistellige) Assoziation (engl. binary association):
- setzt Objekte von genau zwei Klassen zueinander in Beziehung
- UML-Darstellung:
 - Einfache Linie zwischen den Klassen.
 - Name (Bedeutung) der Assoziationsbeziehung (an der Mitte der Linie notiert) und Leserichtung des Namens (ausgefülltes Dreieck).
 - Anzahlangaben (auch: Multiplizitätsangaben, Vielfachheit): mit wie vielen Objekten der gegenüberliegenden Assoziationsseite ist je ein Objekt der Ausgangsseite mindestens/höchstens verbunden.
 - Rollennamen: bezeichnen die Bedeutung der beteiligten Klassen bzw. ihrer Objekte.
 - Assoziationen können uni- oder bidirektional sein (Pfeilspitze an Linie), legt die Navigierbarkeit der Assoziation fest.



Assoziationen


Assoziationen werden zu Attributen von an der Assoziation beteiligten Klassen.

```
public class BibMitglied {
    ...
    // --- Attribute ---
    private AusleihVorgang[] av;
    private String name;
    private String privAnschrift;
    private int mitgliedsNr;
    ...
}

public class AusleihVorgang {
    ...
    // --- Attribute ---
    private BibMitglied ausleiher;
    private Buch leihObjekt;
    private Datum leihBeginn;
    private Datum leihEnde;
    ...
}

public class Buch {
    ...
    // --- Attribute ---
    private String autor;
    private String name;
    private int erscheinungsJahr;
    private String verlag;
    private int buecherNr;
    ...
}
```

Multiplizitäten (engl. multiplicity)

- Multiplizitäten geben an, wie viele Objekte der an der Assoziation beteiligten Klassen jeweils miteinander in Beziehung stehen können.
- Bei zweistelligen Beziehungen: 
- Betrachte Assoziation zwischen Klassen K1 und K2, wobei jedes Ende der Assoziation eine Multiplizität hat.
- Multiplizität c2 drückt für jeden Zeitpunkt t die Anzahl jener Objekte von K2 aus, die mit genau einem Objekt von K1 in Beziehung stehen müssen/dürfen.
- ☐ Entsprechendes gilt analog für Multiplizität c1

Assoziationen mit 1:1-Multiplizität in Java

Fall 1: unidirektional, gerichtet



```
public class Klasse1 {
    ...
    private Klasse2 k2;
    ...
}
```

```
public class Klasse2 {
    ...
    // keine Referenz
    // auf Klasse1-Objekt
}
```

Fall 2: bidirektional, ungerichtet



```
public class Klasse1 {
    ...
    private Klasse2 k2;
    ...
}
```

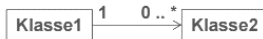
```
public class Klasse2 {
    ...
    private Klasse1 k1;
    ...
}
```

Verknüpfung erfolgt im
Konstruktor oder mit Setter

Assoziationen mit 1:n-Multiplizität in Java

Hinweis: „n“ bedeutet potentiell „viele“: zur Verwaltung von mehreren Objekten gleichen Datentyps kennen wir bislang nur Arrays. Später folgen sog. Container-Klassen, die analog verwendet werden können.

Fall 1: unidirektional, gerichtet



Methoden zum Hinzufügen von Elementen zum Array etc.

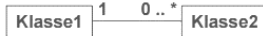
```

public class Klasse1 {
    ...
    private Klasse2[] k2;
    ...
}
  
```

```

public class Klasse2 {
    ...
    // keine Referenz
    // auf Klasse1-Objekt
}
  
```

Fall 2: bidirektional, ungerichtet



```

public class Klasse1 {
    ...
    private Klasse2[] k2;
    ...
}
  
```

```

public class Klasse2 {
    ...
    private Klasse1 k1;
    ...
}
  
```

Assoziationen mit n:m-Multiplizität in Java

Hinweis: „n, m“ bedeutet potentiell „viele“: zur Verwaltung von mehreren Objekten gleichen Datentyps kennen wir bislang nur Arrays. Später folgen sog. Container-Klassen, die analog verwendet werden können.

Fall 1: unidirektional, gerichtet



Methoden zum Hinzufügen von Elementen zum Array etc.

```

public class Klasse1 {
    ...
    private Klasse2[] k2;
    ...
}
  
```

```

public class Klasse2 {
    ...
    // keine Referenz
    // auf Klasse1-Objekt
}
  
```

Fall 2: bidirektional, ungerichtet



```

public class Klasse1 {
    ...
    private Klasse2[] k2;
    ...
}
  
```

```

public class Klasse2 {
    ...
    private Klasse1[] k1;
    ...
}
  
```

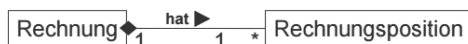
Aggregation (engl. aggregation)

- Aggregation: Spezialform der Assoziation
- Wie Assoziation Beziehung zwischen Klassen.
- Modelliert eine „Teil-Ganzes-“, „enthält-“ oder „hat-Beziehung“.
- UML-Darstellung: unausgefüllte Raute am Beziehungsende auf der Seite des Behälters/des Ganzen.
- Beispiele:



Komposition (engl. composition, composite aggregation)

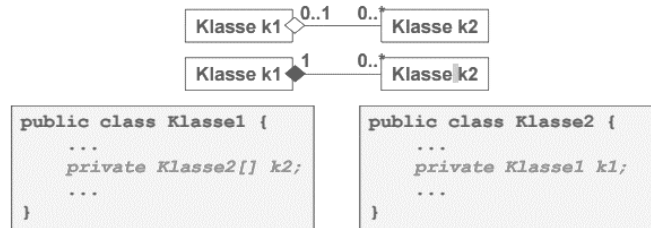
- Komposition: Spezialform der Aggregation
- Aggregation, bei der ein Bestandteil genau zu einem Ganzen gehört und nicht ohne das Ganze existieren kann.
- UML-Darstellung: ausgefüllte Raute am Beziehungsende auf der Seite des Behälters / des Ganzen
- Beispiel



- Semantik:
 - Eine Rechnungsposition gehört immer zu einer Rechnung.
 - Wird die Rechnung gelöscht, werden auch alle davon existenzabhängigen Teile gelöscht.

Aggregation und Komposition in Java

- Wie Assoziation, mit einigen Ergänzungen.
- Aggregation:
 - Für konkretes Objektpaar darf k1 und/oder k2 gleich null sein.
 - Typisch, dass das Ganze stellvertretend für seine Teile handelt.
 - Beispiel: Methode print() der Klasse BibMitgliederVerwaltung durchläuft alle verwalteten BibMitglieder und ruft jeweils deren print()-Methode auf.
- Komposition:
 - k2 darf gleich null sein, k1 muss stets ungleich null sein.



Einschub: String-Vergleich

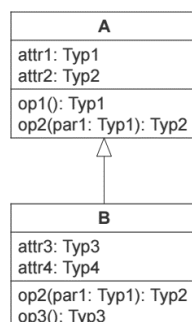
- Für den Vergleich von Zeichenketten stellt die Java-API in der Klasse String folgende Methode bereit: boolean equals(Object anObject) vergleicht den gegebenen String mit dem spezifizierten Objekt.
- Ergebnis ist true, dann und nur dann, wenn das Argument nicht null und ein String-Objekt ist, das dieselbe Sequenz von Zeichen repräsentiert wie das gegebene Objekt.
- Beispiel:

```

String s1 = new String("hellostudents");
String s2 = new String("hellostudents");
String s3 = "hello" + "students";
boolean istGleich;
istGleich = s1.equals(s2); // istGleich: true
istGleich = s1 == s3; // istGleich: false
istGleich = s1.equals(s3); // istGleich: true
  
```

Vererbung

- Mechanismus, der es ermöglicht, sog. Unterklasse(n) aus einer gegebenen Klasse abzuleiten.
- Modelliert eine „ist ein“-Beziehung, z. B. Mitarbeiter ist ein BibMitglied, Buch ist ein Medium, Hund ist ein Tier, Informatik ist ein Studienfach, ...
- Unterklasse erbt dann von der (Ober-) Klasse die Attribute und Methoden.
- Es können weitere Attribute und Methoden hinzukommen, die der Spezialisierung dienen (Spezialisierungsbeziehung zwischen den beteiligten Klassen).
- Aus Unterklasse können weitere Klassen abgeleitet werden, wodurch sich sog. Vererbungshierarchie ergibt.
- Darstellung im UML-Klassendiagramm
 - Linie von Unterklasse zur Oberklasse mit unausgefülltem Dreieck als Pfeilspitze auf der Seite der Oberklasse.
- Definition der Bestandteile (= Attribute und Methoden) der Unterklasse B einer Oberklasse A (Auswahl):
 - Neue Bestandteile von B werden in B
 - selbst festgelegt. Hier: attr3, attr4, op3
 - Abgeleitete Bestandteile von B werden von der Oberklasse A übernommen (Verhaltensgleichheit). Hier: attr1, attr2, op1, op2
 - Überschriebene Bestandteile (nur Methoden) entstehen, wenn die Unterklasse B eine Methode mit genau derselben Signatur wie in der Oberklasse A definiert (und eine neue Implementierung vorliegt). Das ist Spezialisierung im engeren Sinn. Hier: op2



Ober- und Unterklasse

- Eine Oberklasse (auch: Superklasse, engl. super class) beinhaltet die Grundlage für alle von ihr abgeleiteten Unterklassen und ist eine Verallgemeinerung aller ihrer Unterklassen.
 - Beispiel: BibMitglied ist Oberklasse (Superklasse) von / ist eine Generalisierung von BibMitarbeiter
- Eine Unterklasse (auch: Subklasse, engl. sub class) erbt die Attribute und Methoden der Oberklasse, beinhaltet aber noch zusätzliche oder veränderte Eigenschaften der Oberklasse. Die Unterklasse ist eine Spezialisierung der Oberklasse.
 - Beispiel: BibMitarbeiter ist Unterklasse von / ist abgeleitet aus / erbt von / ist eine Spezialisierung von BibMitglied

Die „oberste Oberklasse“: Klasse Object

- In Java ist jede Klasse automatisch Unterklasse der vordefinierten Klasse Object.
- Klasse Object stellt vordefinierte Methoden zur Verfügung, die somit allen Objekten zur Verfügung stehen (gilt auch für Arrays).
- Es kann sinnvoll sein, diese Methoden in Unterklassen passend zu überschreiben.
- Beispiel: toString() liefert textuelle Repräsentation des Objekts

Vererbung in Java

- Vererbung mittels extends bei der Klassen-Definition
- jede Klasse erbt von genau einer anderen Klasse
 - falls nicht explizit angegeben, ist die Oberklasse Object
 - alle Klassen in Java erben direkt oder indirekt von Object
- Einfachvererbung führt zu einer sog. Mono-Hierarchie
- Zugriff auf Methoden oder Attribute der Oberklasse: super
 - notwendig, falls Unterklasse Teile der Oberklasse verdeckt
- falls die Oberklasse keinen Standard-Konstruktor hat:
 - expliziter Aufruf eines Oberklasse-Konstruktors in jedem Konstruktor der Unterklasse notwendig

Vererbung: Beispiel

```
class Form {
    protected int x ;
    protected int y ;
    public Form ( int x , int y ) {
        this . x = x ; this . y = y ;
    }
}
class Rechteck extends Form { // jedes Rechteck ist auch eine Form
    protected int breite ;
    protected int hoehe ;
    public Rechteck ( int x , int y , int breite , int hoehe ) {
        super ( x , y ); // Aufruf des Konstruktors der Oberklasse
        this . breite = breite ; this . hoehe = hoehe ;
    }
}
class Kreis extends Form { // jeder Kreis ist auch eine Form
    protected int radius ;
    public Kreis ( int x , int y , int radius ) {
        super ( x , y ); // Aufruf des Konstruktors der Oberklasse
        this . radius = radius ;
    }
}
```

Abstrakte Klassen

- abstrakte Klasse:
 - kann nicht instanziiert werden
 - es können also keine Objekte direkt von dieser Klasse erzeugt werden,
 - sondern nur von (nicht-abstrakten) Unterklassen dieser Klasse
 - kann als statischer Typ einer Referenz-Variable verwendet werden
- in Java: Deklaration einer abstrakten Klasse mittels abstract
- Beispiel

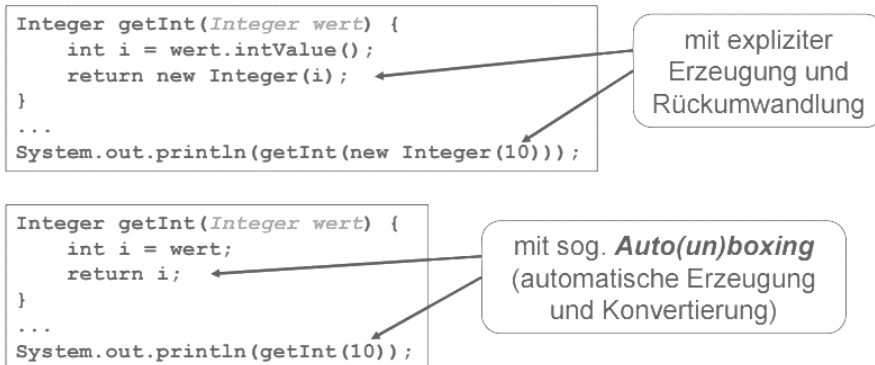
```
abstract class Form { /* ... */ }
class Rechteck extends Form { /* ... */ }
class Kreis extends Form { /* ... */ }
```

Abstrakte Methoden

- abstrakte Klassen können abstrakte Methoden deklarieren
 - legen lediglich die Signatur fest, nicht die Implementierung
- alle Unterklassen der abstrakten Oberklasse...
 - müssen diese abstrakten Methoden implementieren, oder
 - sind selbst abstrakt

Einschub: Wrapper-Klassen

- Viele Methoden (z. B. auch in Java eingebaute) erwarten Parameter vom Typ Object.
- Um diesen Methoden Werte eines primitiven Datentyps zu übergeben, stellt Java für jeden Datentyp eine sog. Wrapperklasse zur Verfügung, die diesen Wert kapselt.
- Wrapper-Klassen: Boolean, Byte, Character, Double, Float, Integer, Long, Short
- Jede Wrapper-Klasse besitzt u. a. eine Methode, um den primitiven Wert zurückzugewinnen.
- Vorgang des Ver- bzw. Entpackens: Boxing bzw. Unboxing
- Beispiel:
 - `Double d = new Double(1.0);`
 - `double dv = d.doubleValue();`
- Wrapper-Objekte als Parameter in Methoden



- Weiterer Vorteil von Wrapper-Klassen: können zusätzliche nützliche (Klassen-) Methoden bereitstellen, wie z. B. `Double.isNaN(...)` (s.o.), die die primitiven Datentypen nicht haben.

Polymorphismus (engl. polymorphism)

- Polymorphie = Fähigkeit, verschiedene Gestalt anzunehmen
- bedeutet, dass eine Erscheinung in vielfacher Gestalt auftritt
- Beispiel: polymorphe Methoden (engl. polymorphic methods)
 - Methoden haben gleichen Namen, tun aber etwas Unterschiedliches.
 - Beispiel (sog. Überladen):
 - Addition + ist für Datentypen int und float definiert
 - für jeden Datentyp eigene Operation
 - gleicher Name aus Komfortgründen
- Java: Deklaration polymorpher Methoden durch
 - Überladung von Methoden
 - Überschreibung von Methoden

Überladung einer Methode (engl. method overloading)

- Mehrfachdefinition von Methoden
- liegt vor, wenn in einem Programmstück mindestens zwei Methodendeklarationen sichtbar sind, die
 - denselben Namen haben und
 - Parameterlisten haben, in denen sich die Datentypen der Parameter in der aufgelisteten Reihenfolge an mindestens einer Stelle oder die sich in der Parameteranzahl unterscheiden.
 - Der Resultattyp spielt in Java beim Überladen keine Rolle.
- Auch ein Konstruktor kann überladen werden.
- Überladen von Methoden ist möglich innerhalb einer Klasse oder innerhalb einer Klasse und einer Unterklasse.
- Beispiel: `System.out. ...`
 - `println(long), println(float), println(Object), ...`
 - Je nach Typ des Arguments `arg` im Aufruf `println(arg)` wird die „passende Version“ der Methode `println` aufgerufen

Einschub: Methodenauswahl in Java

- Finden der anwendbaren und zugreifbaren Methoden
 - anwendbar:
 - Suche in angegebener Klasse und deren Oberklassen
 - Argumentanzahl = Parameteranzahl
 - „Method Invocation Conversion“: kann der Typ jedes Arguments in den Typ des formalen Parameters konvertiert werden?
 - zugreifbar:
 - kann Methode gemäß Modifikator public, private aufgerufen werden? (Später auch: protected und ohne Modifikator).
- Auswahl der spezifischsten Methode

Überschreibung einer Methode (engl. method overwriting)

- liegt vor, wenn es zu einer Methode, die in einer Klasse deklariert ist, eine Methode in einer Unterklasse gibt, die
 - denselben Namen und
 - dieselbe Parameterliste hinsichtlich der Parametertypen in der Reihenfolge der Liste und
 - denselben Resultattyp (oder einen zuweisungskompatiblen engeren Typ davon) hat.
- Anwendung:
 - Objekt der Oberklasse: Verwendung der Methode der Oberklasse
 - Objekt der Unterklasse: Verwendung der Methode der Unterklasse
- Beispiel 1: Methode print() im Bibliotheksbeispiel
 - BibMitglied: print() gibt Daten des BibMitglieds aus.
 - BibMitarbeiter: print() gibt Daten des BibMitglieds zusätzlich zu Daten des BibMitarbeiters aus
- Beispiel 2:
 - Klasse Object hält eine Standardimplementierung von toString() bereit, die (bei Bedarf) in eigenen Klassen überschrieben werden kann.
 - Je nach Typ eines Objekts obj wird bei obj.toString() unterschiedlicher Programmtext ausgeführt.
- Hinweis:
 - Auf eine überschriebene Instanzmethode der Oberklasse kann im Inneren der überschreibenden Klasse per super zugegriffen werden.
 - Ein Aufruf von außen ist nicht möglich

Überladen vs. Überschreiben

- Überladene Methoden
 - haben unterschiedliche Signaturen,
 - haben im Allgemeinen unterschiedliche Implementierungen.
 - Vorkommen:
 - in einer Klasse und einer ihrer Unterklassen oder
 - in einer Klasse (nebeneinander).
 - Klassenmethoden können überladen werden.
- Überschriebene Methoden
 - haben dieselben Signaturen (bis auf evtl. engeren Resultattyp)
 - Klassenmethoden können keine Instanzmethoden überschreiben
 - umgekehrter Fall auch nicht
 - haben im Allgemeinen unterschiedliche Implementierungen.
 - Vorkommen:
 - in einer Klasse und einer ihrer Unterklassen.
 - Klassenmethoden können nicht überschrieben werden (aber „verdeckt“ werden).

Klasse vs. Typ

- Typ
 - Eigenschaft von Variablen und Ausdrücken,
 - definiert Mindestforderung bzgl. anwendbarer Operationen,
 - rein syntaktisch festgelegt (statisch ermittelbar).
- Klasse
 - stellt Konstruktor(en) für Objekte bereit (nicht für abstrakte Klassen),
 - definiert Signatur oder Implementierung der Operationen.
 - Objekt gehört zu der Klasse, mit deren Konstruktor es konstruiert wurde.
 - Mit der Klassendefinition wird ein gleichnamiger Typ eingeführt.
- Hinweis
 - Variablen gleichen Typs können Objekte unterschiedlicher Klassenzugehörigkeit referenzieren (benennen).
 - Sie werden sich im Allgemeinen unterschiedlich verhalten, z. B. Oberklasse A mit zwei Unterklassen A1 und A2. Variable kann mit Typ A deklariert werden und kann Objekte aus A, A1 oder A2 referenzieren.

Polymorphe Variablen in typsicheren Sprachen

- Sei v eine Variable mit Referenzsemantik vom Typ T bzw. der Klasse T.
- Dann dürfen der Variablen v Verweise auf Objekte der Klasse T oder einer beliebigen Unterklasse von T zugewiesen werden.
- Erklärung: Unterklasse verfügt über alle Eigenschaften der Oberklasse und kann daher deren Platz einnehmen
- Typsicherheit: Es dürfen nur Methoden aufgerufen werden, die schon beim statischen Typ von m (Oberklasse Medium) verfügbar sind.

```
// statischer Typ von m: Medium
Medium m;
// dynamischer Typ von m: Hoerbuch
m = new Hoerbuch("Petra Panisch", "Das Super-Mega-Drama", 2007,
                "Dramaqueens", 8, 495);

// Typsicherheit:
// alle Methoden von Medium koennen auf m aufgerufen werden
m.print();           // die anzuwendende Implementierung der Objektmethode
                    // wird zur Laufzeit ausgewaehlt
```

- Eine polymorphe Variable (engl. polymorphic variable) kann im Laufe der Ausführung eines Programms Referenzen auf Objekte verschiedener Klassen haben. Eine polymorphe Variable hat einen
 - statischen Typ: wird durch Angabe der Klasse bei der Deklaration angegeben und kann bei der Übersetzung überprüft werden,
 - dynamischen Typ: wird durch die Klasse des Objekts angegeben, auf den die Variable zur Laufzeit zeigt

Dynamische und statische Bindung

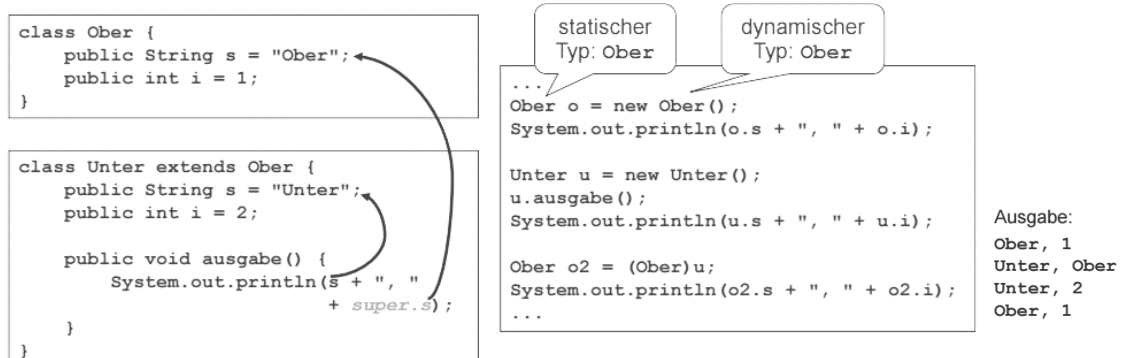
- Klassenhierarchien:
 - Variable, die vom Typ der Oberklasse deklariert ist, kann auch auf Instanzen von deren Unterklassen verweisen.
 - Dort sind eventuell Methoden der Oberklasse überschrieben.
- Kandidaten für Methodenaufruf des in der Variablen referenzierten Objekts:
 - Methode der Oberklasse
 - überschriebene Methodenversion aus der Unterklasse
- Dynamische Bindung (bei Instanzmethoden): zur Laufzeit wird in Abhängigkeit von der Tatsache, ob das Objekt eine Instanz der Ober oder einer Unterklasse ist, die jeweilige Version der Methode aufgerufen.
- sog. virtueller Methodenaufruf
- Ermöglicht so die volle Flexibilität von Vererbung.
- Statische Bindung (bei Klassenmethoden): es wird immer die Methode des Typs genutzt, als der die Variable deklariert ist

Verdecken von (Klassen-) Attributen und Klassenmethoden

- bedeutet, dass
 - ein in der Unterklasse deklariertes Attribut oder Klassenattribut denselben Namen aufweist, wie ein entsprechendes (Klassen-) Attribut der Oberklasse,
 - eine in der Unterklasse deklarierte Klassenmethode dieselbe Signatur (gleicher Name, gleiche Parameterliste (bzgl. Reihenfolge und Typen der Parameter) und gleicher Resultattyp) aufweist, wie eine entsprechende Klassenmethode der Oberklasse.
- Hierbei wird innerhalb der Unterklasse
 - das Attribut oder Klassenattribut der Oberklasse durch das namensgleiche Attribut der Unterklasse verdeckt,
 - die Klassenmethode der Oberklasse durch die signaturgleiche Klassenmethode der Unterklasse verdeckt.
- Regel: beim Zugriff auf (Klassen-) Attribute und Klassenmethoden ist immer der statische Typ an der Zugriffsstelle der relevante

Verdecken von Attributen am Beispiel

Die in der Unterklasse deklarierten Attribute ergänzen die Attribute der Oberklasse; sie kommen zusätzlich hinzu.



(Explizite) Typkonvertierung (engl. casting)

Prüfe mit instanceof, zu welcher Klasse ein konkretes Objekt gehört.

```
Uhr c;
...
if (c instanceof DigitalUhr) {
    ...
}
```

- Im Inneren der if-Anweisung hat c immer noch den statischen Typ Uhr.
- Methoden, die nur die Unterklasse DigitalUhr anbietet, können wegen der geforderten Typsicherheit nur umständlich aufgerufen werden; Abhilfe: explizite Typwandlung

```
Uhr c;
...
if (c instanceof DigitalUhr) {
    DigitalUhr digital = (DigitalUhr)c;
    ...
}
```

Schnittstellen

- Mit einer Schnittstelle (eingeleitet durch Schlüsselwort: interface) wird in Java definiert, welche Methoden eine diese Schnittstelle implementierende Klasse (Schlüsselwort: implements) mindestens haben muss.
- In einem Interface können deklariert werden:
 - Konstanten und
 - Signatur(en) von Methode(n); Die Implementierung der angegebenen Methode(n) muss in einer Klasse erfolgen.
 - Alles andere ist nicht zulässig.
- In einer Interfacedeklaration haben
 - Konstantendeklarationen automatisch die Modifizierer static und final und public,
 - Methodendeklarationen automatisch die Modifizierer abstract und public.
 - Deshalb brauchen diese nicht angegeben zu werden.
- Typ einer Objektvariablen kann nun auch eine Schnittstelle sein (statt bisher: Klasse).
 - Referenz kann aber nur auf ein Objekt verweisen.
 - Wird ihr ein Objekt zugewiesen, so muss dessen Klasse die Schnittstelle implementieren.
- Vererbung zwischen Schnittstellen mit extends
- Mehrfachvererbung von Schnittstelle ist möglich:

```
interface Inter3 extends Inter1, Inter2 {
    ...
}

class Demo implements Inter1, Inter2 {
    ...
}
```

- Auflösung/Behandlung potentieller Mehrdeutigkeiten bei Mehrfachvererbung bei Schnittstellen:
 - Methoden: Werden zwei gleich benannte Methoden über verschiedene Pfade ererbt,
 - so wird bei identischen Signaturen eben diese Signatur übernommen,
 - so werden bei unterschiedlichen Signaturen beide übernommen und dann als überladene Methoden behandelt.
 - Konstanten:
 - Werden zwei gleich benannte Konstanten über verschiedene Pfade geerbt, meldet der Übersetzer Fehler, (nur) wenn die Konstante verwendet wird.
 - Ein Interface kann eine Konstante deklarieren, die eine/mehrere geerbte Konstanten gleichen Namens verdeckt.

Pakete

- Paket (engl. package): Zusammenstellung der Vereinbarungen von als zusammengehörig betrachteten Java-Klassen und Java-Interfaces.
- Paket besitzt einen Namen, der – wie von Dateiverzeichnissen her bekannt – hierarchisch aufgebaut sein kann:
 - java.lang
 - com.apple.quicktime.v7
- Paketname definiert Namensraum für die im Paket vorkommenden Klassendeklarationen und sollte nur aus Kleinbuchstaben bestehen.
- Hinweis:
 - Daher prinzipiell/theoretisch möglich, innerhalb eines Pakets eine neue Klasse String zu realisieren.
 - Eindeutigkeit erfolgt durch Verwendung qualifizierter Klassennamen (java.lang.String bzw. <mein_paket>.String).

Paket uhren: Interface Taktgeber und Klasse TimeZone

```
package uhren;
interface Taktgeber {
    long leseZeitInSekunden();
}
```

```
package uhren;
public class TimeZone {
    private String identifikator;
    public TimeZone(String id) {
        setzeIdentifikator(id);
    }
    public void setzeIdentifikator(String id) {
        identifikator = id;
    }
    ...
}
```

Klasse Uhr

```
package uhren;
public abstract class Uhr implements Taktgeber {
    protected static long zaehler = 0;
    protected long seriennummer;
    protected long minuten;
    protected long stunden;
    protected TimeZone zeitzone = new TimeZone("UTC");
    protected Uhr() {
        seriennummer = zaehler++;
    }
    protected Uhr(long min) {
        this();
        stunden = min / 60;
        minuten = min % 60;
    }
    public long leseSeriennummer() {
        return seriennummer;
    }

    public abstract void anzeigen();
    public void setzeZeit(long std, long min) {
        stunden = std; minuten = min;
    }
    public void setzeZeit(long std) {
        setzeZeit(std, 0);
    }
    public long leseZeitInSekunden() {
        return (stunden * 3600) + (minuten * 60);
    }
    public void setzeZeitzone(TimeZone tz) {
        if (tz == null) {
            return;
        }
        zeitzone = tz;
    }
    public TimeZone leseZeitzone() {
        return zeitzone;
    }
    ...
}
```

Zugriff auf Klassen/Schnittstellen eines Pakets von Außerhalb

- Um außerhalb eines Pakets eine darin deklarierte Klasse oder Schnittstelle zu verwenden, gibt es zwei Möglichkeiten:
- Zugriff auf eine Klasse in einem Paket über den qualifizierten Namen <Paketname>.<Klassenname>
- Alternative:
- Paketname wird mittels import bekannt gemacht.
- Dann kann auf eine im Paket enthaltene Klasse mittels <Klassenname> zugegriffen werden

Klassenbibliothek (engl. library)

- Sammlung vordefinierter, häufig verwendeter Klassen, auf die bei der Programmierung zugegriffen werden kann.
- Java: jeweils Menge von Paketen
- Java selbst liefert z. B. die Pakete:
 - java.io Ein-/Ausgabe in Dateien o. ä.
 - java.lang zentrale Klassen von Java (z. B. Object)
 - Alles in java.lang.* wird automatisch importiert.
 - java.net Verwaltung von Netzwerkverbindungen
 - java.sql Datenbankanbindung
 - java.util Nützliche Klassen (z. B. Kalender, Listen)
 - javax.swing graphische Oberfläche

Methoden ausgewählter Bibliotheksklassen

- Klasse java.lang.System stellt die Schnittstelle einer laufenden Java-Applikation zur Systemumgebung bereit.
- Zunächst am wichtigsten: Standard-Ein/Ausgabe, über die Zeichen mit der Shell ausgetauscht werden können:
 - static `PrintStream out` The "standard" output stream.
 - static `PrintStream err` The "standard" error output stream.
 - static `InputStream in` The "standard" input stream.
- Wichtigste Methoden von java.lang.`PrintStream`:
 - void `print(String s)` Print a string.
 - void `println()` Terminate the current line by writing the line separator string.
 - void `println(String x)` Print a String and then start a new line.

Konstruktoren der Klasse java.lang.String

- `String()` Initializes a newly created String object so that it represents an empty character sequence.
- `String(char[] value)` Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
- `String(char[] value, int offset, int count)` Allocates a new String that contains characters from a subarray of the character array argument.
- `String(byte[] bytes)` Constructs a new String by decoding the specified array of bytes using the platform's default charset.
- `String(byte[] bytes, String charsetName)` Constructs a new String by decoding the specified array of bytes using the specified charset.
- `String(byte[] bytes, int offset, int length) ...`
- `String(byte[] bytes, int offset, int length, String charsetName) ...`
- `String(String original)` Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string
- `char charAt(int index)` Returns the character at the specified index.
- `int compareTo(String anotherString)` Compares two strings lexicographically (returns 0 if equal).
- `int compareToIgnoreCase(String str)` Compares two strings lexicographically, ignoring case differences.
- `boolean endsWith(String suffix)` Tests if this string ends with the specified suffix.
- `boolean equalsIgnoreCase(String anotherString)` Compares this String to another String, ignoring case considerations.
- `int indexOf(int ch)` Returns the index within this string of the first occurrence of the specified character.
- `int lastIndexOf(String str)` Returns the index within this string of the rightmost occurrence of the specified substring.
- `String replace(char oldChar, char newChar)` Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
- `String substring(int beginIndex, int endIndex)` Returns a new string that is a substring of this string.
- `String toUpperCase()` Converts all of the characters in this String to upper case using the rules of the default locale

Bibliotheksklasse java.lang.Math

- Klasse java.lang.Math enthält Methoden, die mathematische Funktionen zur Verfügung stellen.
- Methoden dieser Klasse sind statische Methoden.
- Sie können verwendet werden, indem dem Methodennamen Math. vorangestellt wird.

Konstanten und Methoden der Klasse Math (Auswahl)

- static double `E` The double value that is closer than any other to e, the base of the natural logarithms.
- static double `PI` The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.
- static double `abs(double a)` Returns the abs. value of a double value.
- static float `abs(float a)` Returns the abs. value of a float value.
- static int `abs(int a)` Returns the absolute value of an int value.
- static long `abs(long a)` Returns the absolute value of a long value.
- static double `sin(double a)` Returns the trigonometric sine of an angle.

Konstanten und Methoden der Klasse Math (Auswahl)

- static double exp(double a) Returns Euler's number e raised to the power of a double value.
- static double log(double a) Returns the natural logarithm (base e) of a double value.
- static double max(double a, double b) Returns the greater of two double values.
- static double pow(double a, double b) Returns the value of the first argument raised to the power of the second argument.

Beispiel statischer und dynamischer Typ Übung

```
interface Sammlerstueck { /* ... */ }
class Oldtimer implements Sammlerstueck { /* ... */ }
class Briefmarke implements Sammlerstueck { /* ... */ }
// =====
Sammlerstueck objekt = null ;
    // stat . Typ: Sammlerstueck , dyn. Typ: keiner
objekt = new Oldtimer ();
    // stat . Typ: Sammlerstueck , dyn. Typ: Oldtimer
objekt = new Briefmarke ();
    // stat . Typ: Sammlerstueck , dyn. Typ: Briefmarke
```