Skript
Informatik B

Objektorientierte Programmierung in Java

Sommersemester 2011

Auf eine Unterscheidung der männlichen und weiblichen Form wird zur Vereinfachung der Lesbarkeit verzichtet. Wo die weibliche oder männliche Form verwendet wird, ist automatisch immer auch die jeweilig andere Form gemeint.

Die Unterlagen verwenden im Allgemeinen deutsche Begriffe. Wichtige englische Begriffe werden in Klammern zusätzlich eingeführt und sollten in der englischsprachigen Welt der Informatik ebenfalls gelernt werden. Wo die Verwendung englischer Begriffe auch im deutschsprachigen Raum üblich ist, werden auch wir den englischen Fachbegriff einsetzen.

Organisatorisches

Wichtige Informationen:
- Vorlesungs- und Übungsunterlagen: http://www-lehre.inf.uos.de/~binf/
- https://list.serv.uni-osnabrueck.de/mailman/listinfo/binf11
- StudIP http://studip.serv.uos.de
- Literatur auf der Webseite

Ablauf:
- Vorlesung mit begleitenden Übungen (4+2 SWS, 9 ECTS).
- Übungsgruppen: Besprechung der Lösungen und Vorbesprechung des neuen Übungsblatts mit dem Übungsleiter.
- Testatgruppen: Die Kleingruppen (in der Regel Zweiergruppen) stellen ihre Lösung einer Tutorin bzw. einem Tutor während eines Testattermins vor.
- Bewertung durch die Tutorin bzw. den Tutor während des Testattermins und auf Basis der vorher erfolgten Abgaben.

Empfohlene Begleitliteratur

Stukturierung der Begleitliteratur:

Gebiet	Lehrbuch / Tutorial (schrittweise Anleitung mit Konzepten)	Referenz / Nachschlage-werke	Werkzeuge
Software Engineering			
OO(++) Konzepte			
UML Sprache und Modellierung			
Programmiersprache Java			

Lehr- und Lernbücher, Tutorials:

[JavaInsel10] Christian Ullenboom: Java ist auch eine Insel
Galileo Press, 9. Auflage, 2010, 1475 S., ISBN: 978-3-8362-1371-4
Inhalt: Java Sprachkonstrukte und Hintergründe
Stufe: Einsteiger, Fortgeschrittene, Experten
e-Book: http://openbook.galileocomputing.de/javainsel9/

[JavaSunTutorial10] Sun: The Java Tutorial, 2010
Inhalt: „How To ..“ Schritt-für-Schritt Erklärung
Stufe: Einsteiger, Fortgeschrittene, Experten
Volltext: http://java.sun.com/docs/books/tutorial
http://download.oracle.com/javase/tutorial/

[Bloch08] Joshua Bloch: Effective Java: Programming Language Guide
Pearson Education Inc., 2. Auflage, 2008, ISBN: 978-0-321-35668-0
Inhalt: Gute Java Codierungspraxis
Stufe: Fortgeschrittene
(auch als Safari Book Online im Universitätsnetz freigeschaltet)

[Oesterreich09] Bernd Oestereich: Analyse und Design mit UML 2.3 - Objektorientierte Softwareentwicklung,
9. Auflag, Oldenbourg Verlag, 2009,
Inhalt: OO Konzepte und UML
Stufe: Einsteiger, Fortgeschrittene

[Gamma95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.:
Design Patterns, Elements of Reusable Object-Oriented Software,
Addison-Wesley, 1995, 416 S., ISBN: 978-0-2016-3361-0
Inhalt: Standardwerk zum Thema Entwurfsmuster

[Freemann04] Eric Freemann, Elisabeth Freemann, Bert Bates, Kathy Sierra und Mike Loukides: Head First Design Patterns
O'Reilly Media; 1. Auflage, 2004, ISBN: 978-0596007126

Nachschlagebücher, Referenzwerke:

[Flanagan05] David Flanagan: Java in a Nutshell
O'Reilly, 5th Edition, 2005, 1254 S., ISBN: 978-0-5960-0773-7
Inhalt: Sprachkonstrukte und Klassenbibliothek
Information: http://www.oreilly.com/catalog/javanut4/
(auch als Safari Book Online im Universitätsnetz freigeschaltet)

[JavaAPISpec10] Sun: Java 2 Standard Edition API Specification, 2010
Inhalt: "offizielle", vollständige Dokumentation der Klassenbibliothek
Standardnachschlagewerk
Volltext: http://java.sun.com/javase/6/docs/api/
http://download.oracle.com/javase/6/docs/api/

[JavaLangSpec10] Sun: Java Language Specification, Third Edition
Inhalt: formale Beschreibung der Sprachsyntax / -konstrukte
Volltext: http://java.sun.com/docs/books/jls/index.html

[Balzert05] Heide Balzert: UML 2 kompakt
Spektrum-Akademischer Vlg, 2. Auflage, 2005,
ISBN: 978-3827413895
Inhalt: UML Referenz

Bücher und Informationen zu begleitenden Werkzeugen:

- Sun: Java SDK Tools and Utilities: http://java.sun.com/j2se/...
- Sun: How to Write Doc Comments for the Javadoc Tool:
  http://java.sun.com/j2se/javadoc/writingdoccomments/
- Eclipse: http://www.eclipse.org/

Inhalt

… wird schrittweise erweitert

Einleitung

Was sollten wir im Kurs “Objektorientierte Programmierung mit Java” lernen?

Sollen wir die Feinheiten der Programmiersprache Java Version x.y lernen?

Wir werden niemals alle Feinheiten lernen können: Es gibt überwältigend viel Material und Wissen zu den verschiedenen Java Versionen. Dies umfasst die Sprachdetails sowie Anwendungserfahrungen.

Wir sollten auch nicht anstreben langfristig alle Feinheiten zu lernen: Der Wert menschlicher Handbücher ist nicht sehr hoch. Zudem ändern sich Feinheiten sehr schnell. Feinheiten sollte man nachlesen, wenn man sie akut benötigt, denn heute gelernte Feinheiten sind mit hoher Wahrscheinlichkeit nicht mehr gültig, wenn sie tatsächlich erst in zwei Jahren gebraucht werden. Zum einen ändert sich eine Programmiersprache von Version zu Version. Zum anderen ändert sich auch die Nachfrage nach der Programmiersprache. Wenn wir heute Java lernen, kann es schnell passieren, dass z.B. Objective-C bald die neue und am häufigsten nachgefragte Sprache ist. Aussagen für die Zukunft sind immer spekulativ.

Der Fokus auf eine bestimmte Sprache oder gar Sprachversion ist also bedenklich.

Warum sollten wir dann Java überhaupt betrachten?

Die objektorientierte Denkweise lernt sich wie alles andere auch am besten an einem konkreten Beispiel. In unserem Fall ist es Java, da diese Sprache die zur Zeit beständigste Sprache mit hoher Praxisrelevanz ist.
Wichtig ist dabei, dass wir nicht nach den Sprachdetails suchen, sondern anhand dieser das beständigere, generischere Wissen dahinter erarbeiten.
Beständiger als ein spezielles Sprachdetail sind die grundlegenden objektorientierten Konzepte. Mit Java lernen wir, wie diese beispielsweise in eine Programmiersprache abgebildet werden können.
Auch können wir daran lernen wie man sich grundsätzlich eine neue Programmiersprache erarbeitet. Dazu gehört auch das Umfeld der Programmiersprache wie z.B. die Entwicklungsumgebung, helfende Werkzeuge, Programmierprinzipien, das Vorgehen zur Fehlerbeseitigung sowie das generelle Entwicklungsvorgehen.
Die Wahrscheinlichkeit, dass Sie in Ihrer beruflichen Zukunft eine andere, Ihnen unbekannte Programmiersprache in sehr kurzer Zeit selbstständig erlernen müssen ist sehr hoch.
Beständiger als ein Sprachdetail ist außerdem das Wissen, wie die objektorientierten Konzepte sinnvoll angewendet werden können. Dies umfasst beispielsweise den Entwicklungsprozess mit grafischen Entwurfssprachen wie z.B. UML (Unified Modeling Language) sowie Erfahrungen zu guten Lösungen, wie sie in sogenannten Entwurfsmustern festgehalten sind.

Objektorientiertes Paradigma

Ein Paradigma ist eine Denkweise und Herangehensweise an ein Problem.

Zur Abgrenzung sollte man wissen, dass das objektorientierte Paradigma nur eines der vier heute wichtigsten Paradigmen ist.

Unterscheidung der Paradigmen:
- Imperativ und Prozedural: Alles wird als eine große, lineare Liste mit Programmbefehlen mit Prozeduraufrufen für verschiedene Teilaufgaben betrachtet. Mit Sprungziel-Anweisungen können Programmteile übersprungen oder wiederholt werden. Beispiele: Cobol, Assembler.
- Funktional: Alles wird als Funktion und Anwendung von Funktionen im mathematischen Sinne betrachtet. Beispiel: LISP, Scheme
- Logisch: Die Welt wird als Fakten und nach Gesetzen der Logik abgeleitetes neues Wissen betrachtet. Diese Wissensableitung nennt sich Inferenz. Beispiel: Prolog
- Objektorientiert: Alles wird als ein Zusammenspiel interagierender Objekte betrachtet, die einen Zustand kapseln und über Nachrichten kommunizieren. Beispiele: Smalltalk, Java

Programmiersprachen unterstützen häufig Konzepte verschiedener Paradigmen (z.B. die Sprachen Oz, C++).
In jeder Programmiersprache kann mit jedem Paradigma entwickelt werden. Theoretisch kann auch mit Assembler objektorientiert programmiert werden und mit Jave imperativ. Eine Programmiersprache unterstützt ein bestimmtes Paradigma, erzwingt es aber nicht.

Begriffe:

Objektorientierung = OO
Objektorientierte Programmierung = OOP
Objektorientierte Analyse = OOA
Objektorientierter Entwurf = OOD
Objektorientierte Modellierung = OOM

Objektorientierung ist ein Paradigma.
Damit ist die Objektorientierung mehr als nur die objektorientierte Programmierung.
Neben der objektorientierten Programmierung findet sich dieses Paradigma auch in anderen Gebieten wie z.B. objektorientierten Datenbanken.
Im Gebiet der objektorientierten Analyse wird die Problemstellung („Was“) auf objektorientierte Art und Weise ermittelt und beschrieben.
Im objektorientierten Entwurf werden objektorientierte Denkweisen und Hilfsmittel eingesetzt, um die Umsetzung und Lösung („Wie“) abzuleiten und zu formulieren.
Die objektorientierte Modellierung befasst sich mit geeigneten Sprachen und Vorgehensprinzipien zur meist grafischen Darstellung von Problem, Entwurf und Lösung.

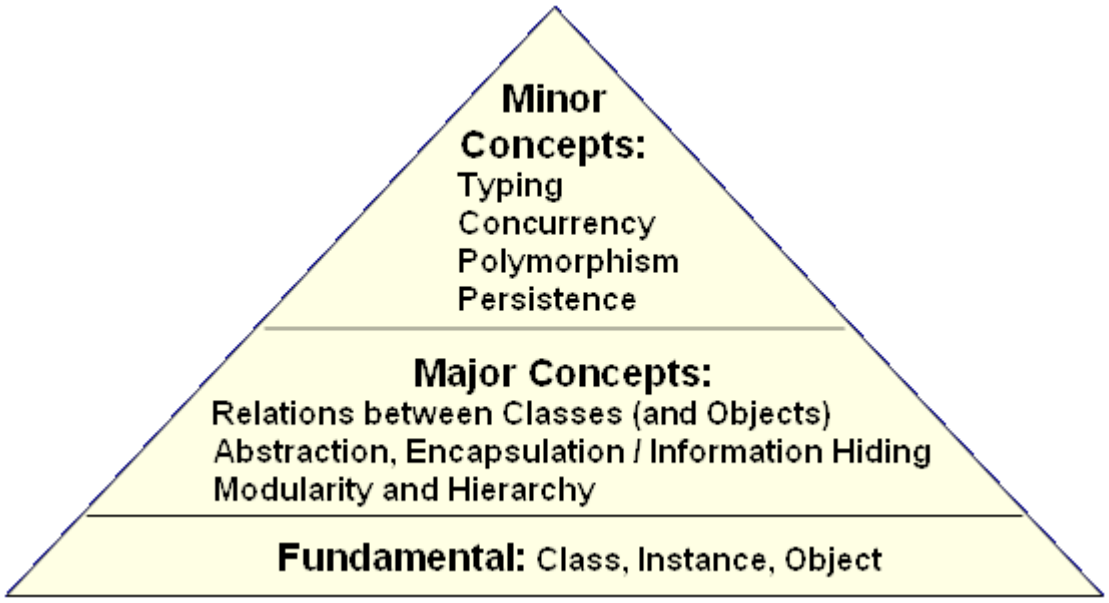
Java ist eine objektorientierte Programmiersprache, d.h. sie unterstützt eine objektorientierte Softwareentwicklung.

Die Programmierung mit einer objektorientierten Programmiersprache führt allerdings nicht automatisch zu einem objektorientierten Programm.

Kapitel 1:
Grundlegende objektorientierte Konzepte

Es muss immer zwischen dem objektorientierten Konzept und der konkreten Umsetzung in einer Sprache (hier Java) unterschieden werden.

Die grundlegenden objektorientierten Konzepte können in fundamentale, wichtige und weniger wichtige unterteilt werden [Boo94]:



- Fundamentale Konzepte: Klasse, Instanz, Objekt
- Wichtige Konzepte: Beziehungen (engl. Relations) zwischen Klassen (und Objekten), Abstraktion (engl. Abstraction), Kapselung (engl. Encapsulation) / Geheimnisprinzip (engl. Information Hiding), Modularität und Hierarchie

- Weniger wichtige Konzepte: Typisierung (engl. Typing), Nebenläufigkeit (engl. Concurrency), Polymorphismus/Polymorphie (engl. Polymorphism), Persistenz (engl. Persistence)

Eine Vorgehensweise oder Sprache ist nur dann wirklich objektorientiert, wenn sie die fundamentalen und weitgehend auch die wichtigen objektorientierten Konzepte umsetzt. Die weniger wichtigen, weiteren Konzepte sind nach dieser Einteilung nicht zwingend in einer objektorientierten Umgebung, finden sich aber häufig in diesem Kontext wieder bzw. ergänzen sich ideal zu den übrigen objektorientierten Konzepten.
Die weniger wichtigen Konzepte sind demnach nicht grundsätzlich weniger wichtig. Sie haben vielmehr speziell in Bezug auf das objektorientierte Paradigma eine weniger charakterisierende Bedeutung

[Boo94] G. Booch: Object-Oriented Analysis and Design, 2nd Edition, Benjamin/Cummings, Redwood City, CA, 1994

1.1 Objekte, Klassen und Instanzen

Konzept Objekt

Im objektorientierten Paradigma wird die Welt bestehend aus Objekten und ihren Interaktionen betrachtet.

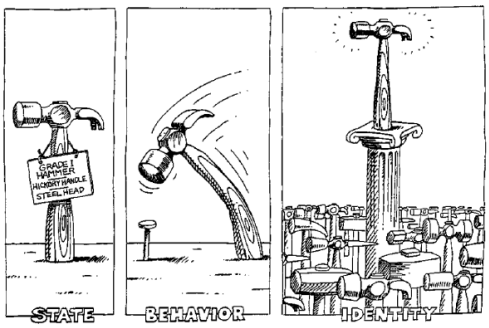
Was ist ein Objekt?
Ein Objekt repräsentiert eine Einheit (engl. Entity), ein Ding der realen Welt

- Physikalisch, Beispiel: ein LKW
- Konzeptuell, Beispiel: ein chemischer Prozess
- In der Software, Beispiel: eine Datenstruktur wie z.B. eine verkettete Liste

Bemerkung zur Programmgestaltung (Modellierung):
In der OO Welt ist ein Objekt ist immer etwas, das klar abgegrenzt werden kann und eine Bedeutung für die Anwendung hat. Objekte, die keine Relevanz für die betrachtete Anwendung haben, sollten nicht modelliert und implementiert werden. Wozu auch?

Eigenschaften eines Objekts:
Ein Objekt hat

- einen Zustand,
- ein Verhalten und
- eine Identität.



[Boo94]

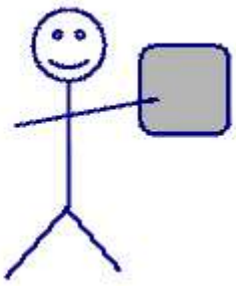
Abbildung der Objekteigenschaften in einer Programmiersprache:

Objekt-eigenschaft	**Repräsentation in einer OO Programmiersprache**	**Beispiel**
Zustand	Mit konkreten Werten belegte Attribute bzw. Instanzvariablen	Ein Objekt vom Typ `Person` hat `name = Bart` und `alter = 10` als mit konkreten Werten belegte Attribute.
Verhalten	Operationen bzw. Methoden	Ein Objekt vom Typ `Person` kann seinen Jahrgang aus dem gespeicherten Alter und dem aktuellen Jahr errechnen (realisiert durch die Methode `jahrgang()`).
Identität	(Nicht so einfach – das schauen wir uns gleich genauer an.)	Zwei Objekte vom Typ `Person`, die beide den Namen `Bart` und das Alter `10` haben, sind trotzdem nicht identisch. Auch zwei Menschen, die Bart heißen, 10 Jahre alt sind und jeweils ihren Jahrgang ausrechnen können, sind dennoch zwei unterschiedliche Individuen.

Die Identität wird in verschiedenen Programmumgebungen unterschiedlich repräsentiert. Häufig wird die Speicheradresse eines Objekts als Hilfsmittel zur Repräsentation der Identität verwendet. Identität und Gleichheit werden wir später noch etwas genauer betrachten.

Bemerkung zur Programmgestaltung (Modellierung):
Die Identität zeigt ein Prinzip sehr deutlich: Objekte in der realen Welt und Objekte im Programm sind zwei verschiedene Denkwelten, die erst aufeinander abgebildet werden müssen. Dabei gibt es nicht immer eine klare und eindeutige Abbildungsvorschrift.
In der Softwareentwicklung wird zunächst auf der Ebene der realen Welt gedacht und modelliert und erst dann auf eine konkrete Programmumgebung (mit ihren Möglichkeiten und Einschränkungen) abgebildet.

Konzept Klasse



Klassen
- sind Baupläne für einzelne konkrete Ausprägungen.
- sind Gruppen von Objekten mit
  - gemeinsamen Eigenschaften (Attributen),
  - gemeinsamem Verhalten (Operationen),
  - gemeinsamen Beziehungen.
- sind Klassifikationen von Objekten.
- haben keine Identität.

Allgemeine Form einer Klasse in der Programmiersprache Java:

```
[Modifikator] class Klassenname [extends Oberklasse] [implements Interface]
{
        Attributdeklarationen
        Methodendeklarationen
}
```

Die Sprachelemente in eckigen Klammern sind optional.

Beispiele für eine Klasse in Java

```
class Person { }
```

Eine Klassenbeschreibung entspricht der Festlegung eines neuen Typs.
Ein Typ legt einen Datenbereich fest sowie Operationen, die auf den Daten dieses Bereichs anwendbar sind.

- Beispiel: Der Typ int legt mit seiner 32-Bit Länge in Java Zahlen im Wertebereich -2^{31} bis $2^{31} - 1$ (–2147483648 bis 2147483) fest. Mit diesen Werten können (nur) bestimmte arithmetische Operationen durchgeführt werden (z.B. Addition).
- Beispiel: Der Typ String legt einen Wertebereich fest, der aus einer Kombination von Zeichen besteht. Erlaubte Operationen darauf sind String-Verarbeitungsoperationen (z.B. Verkettung).

Zum Thema Typen werden wir später noch etwas mehr hören.

Attribut bzw. Instanzvariable: Eigenschaften einer Klasse bzw. eines Objekts
Sie werden innerhalb einer Klasse definiert und zur Laufzeit angelegt, wenn eine Instanz der definierenden Klasse erzeugt wird (siehe weiter unten). Eine Instanzvariable kann in jeder Instanz, die aus der gleichen Klasse erzeugt wurde, einen anderen Wert annehmen. Die Wertebelegungen von Instanzvariablen ändern sich in der Regel im Programmverlauf. Es ergeben sich verschiedene Objektzustände.

Abbildung des Konzepts der Attribute und Instanzvariablen in die Programmiersprache Java:

Allgemeine Java Syntax:

```
[Modifier] Typ einAttributName;
```

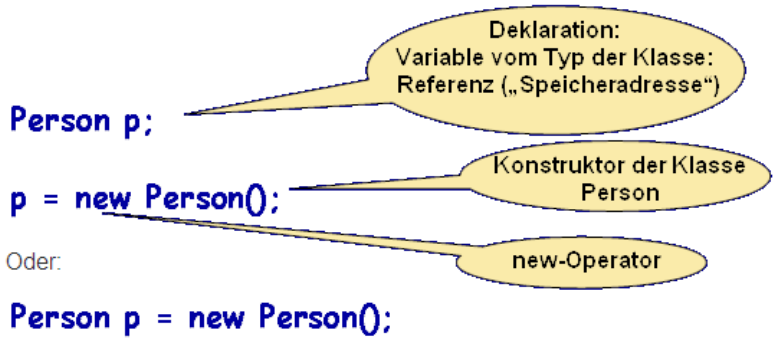
Beispiel:

```
class Person  {
  String name;
  int alter;
  int schuhgroesse;
}
```

In der Klasse Person sind im angegebenen Beispiel keine Operationen vorhanden, sondern lediglich die Attribute (Instanzvariablen).

Bemerkung zur Programmgestaltung (Modellierung):
- Eine Klasse ohne Methoden hat gegebenenfalls keine Berechtigung als eigene Klasse im Programm zu sein.
- Attributnamen sollten per Konvention mit Kleinbuchstaben beginnen und zur besseren Programmlesbarkeit sprechend sein, d.h. die Bedeutung des Attributs widerspiegeln.

Instanziierung: Erzeugung einer neuen Instanz bzw. eines neuen Objekts aus einer Klasse.
Eine Instanz ist damit eine konkrete Ausprägung einer Klasse im Speicher.
Instanzen der gleichen Klasse halten sich an die gleiche Klassendefinition und –deklaration (gleiche Methoden und Attribute).
Beispiel in Java:



In der Deklaration im Beispiel wird eine Variable vom Typ Person angelegt.
Der Operator new erzeugt eine neue Instanz vom Typ Person.
Die Speicheradresse dieser neuen Instanz wird in der deklarierten Variablen abgelegt.
Die Speicheradresse ist die Referenz auf die Instanz im Speicher.

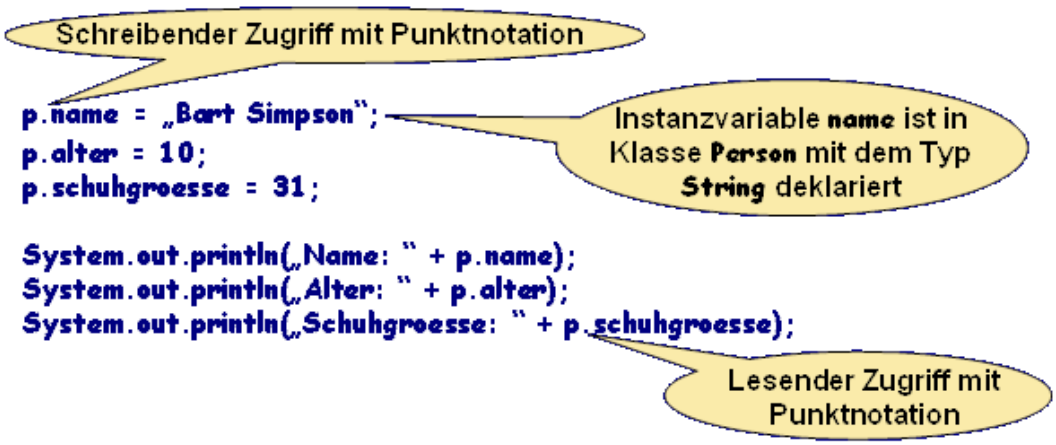
In Java:
Instanzvariablen werden bei Instanzerstellung automatisch mit Standardwerten belegt
Standardwert für eine Referenz: null

Nach dem Anlegen der Instanz im Speicher wird im Beispiel von oben der sogenannte Konstruktor der Klasse Person mit Namen Person() gerufen. Er kann weitere Initialisierungen und Aktionen durchführen, die bei Instanzerzeugung notwendig sind. Über den Konstruktor werden wir später noch mehr sagen.

Hinweis zur Programmierung:
- Zur Vermeidung von Fehlern sollten Instanzvariablen im Konstruktor initialisiert werden. Sich auf Standardwerte zu verlassen bindet das Programm an eine bestimmte Sprache und eine bestimmte Compiler/Interpreterversion. Andere Programmiersprachen kennen z.B. keine Standardwerte oder verwenden andere Standardwerte.
- Es sollten keine unnötigen Instanzen erzeugt werden: Instanzerzeugung und –zerstörung ist teuer und Instanzen kosten Speicher- und Verwaltungsaufwand. Manchmal muss man genau schauen, um zu erkennen, dass unnötige Instanzen erzeugt werden.

Zugriff auf die Attribute:

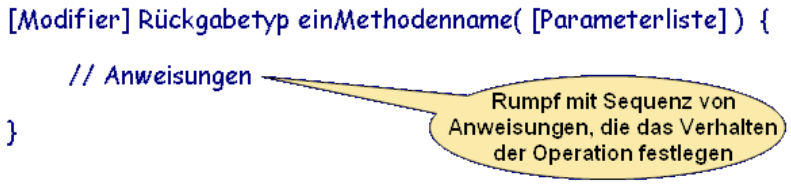
Zugriff in allgemeiner Form in Java: einObjektname.einAttributname
Beispiel:



Hinweis zur Programmgestaltung (Modellierung) und zur Programmierung:
- Der Zugriff auf Klassenelemente über die Punktnotation ist in den meisten objektorientierten Programmiersprachen üblich.
- Attribute sollten nach außen immer versteckt werden und nur über die Operationen des kapselnden Objekts zugegriffen werden (siehe auch unten, Thema Kapselung).

Operationen bzw. Methoden in Klassen:
- definieren das Verhalten von Objekten.
- werden innerhalb einer Klassendefinition deklariert und definiert.
- haben Zugriff auf alle Daten des Objekts.
- sind das Gegenstück zu den Funktionen bzw. Prozeduren in imperativen bzw. prozeduralen Programmiersprachen, sind aber demgegenüber immer an ein Objekt / eine Klasse gebunden und arbeiten nur mit den Daten dieses Objekts und den Methodenparametern.

Allgemeine Java Syntax:



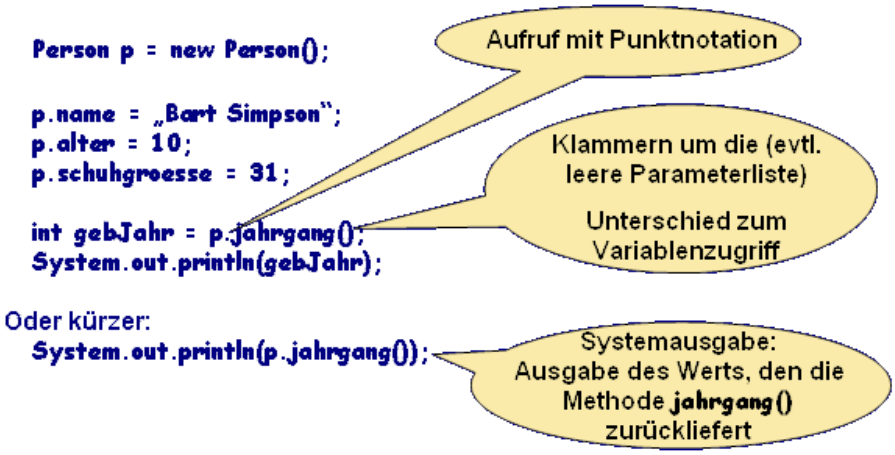
Beispiel:
```
class Person {
        String name;
        int alter;
        int schuhgroesse;

        int jahrgang() {
                return (2011 – alter);      // Rückgabe des berechneten
                                            // Werts vom Typ int
        }
}
```

Begriff Signatur:
- Signatur der Methode = interner Name der Methode.
- Bestandteile der Signatur in Java:
  nach außen sichtbarer Methodenname + Parametertypen + Parameterreihenfolge
- Anhand der Signatur wird zur Laufzeit entschieden, welche Methode aufgerufen werden soll.

Bemerkung:
- In Java ist der Rückgabetyp nicht Teil der Signatur. In anderen Programmiersprachen kann das anders sein.
- Die Methode `jahrgang()` ist nicht sehr schlau gelöst: Jedes Jahr muss der Programmierer die Jahreszahl anpassen (hohe Fehlergefahr!).

Zugriff auf die Methoden bzw. Operationen:

Zugriff in allgemeiner Form in Java: einObjektname.einMethodenname([Parameterwert]);
Beispiel:



Bemerkung zur Terminologie:

- Objekt versus Instanz: Der Begriff *Objekt* bezieht sich in der Tendenz mehr auf die reale Welt und deren Modellierung während der Begriff *Instanz* tendenziell stärker auf Programmebene zu sehen ist. Eine strikte Trennung gibt es allerdings nicht und die Begriffe werden häufig synonym verwendet.
- Attribut versus Instanzvariable: Attribut und Instanzvariable verhalten sich wie die Begriffe *Objekt* und *Instanz*. Eine Variable hat begrifflich noch stärkeren Programmbezug als der Begriff *Instanz*.
- Operation versus Methode: *Operation* und *Methode* verhalten sich in ihrer begrifflichen Verwendung wie die Begriffe *Objekt* und *Instanz*.
- Methode versus Nachricht bzw. Botschaft: Ein Objekt sendet eine Nachricht (oder auch Botschaft) an ein anderes Objekt. Dieses empfängt die Nachricht. Dadurch wird eine Methode im empfangenden Objekt ausgelöst. In der Regel hat die Nachricht den gleichen Namen wie die aufgerufene Methode. Methode und Nachricht repräsentieren allerdings zwei verschiedene Standpunkte (Innensicht und Außensicht).
- Deklaration versus Definition: Eine Deklaration legt ein Programmelement (z.B. eine Variable) mit dem benötigten Speicher an. Die Definition füllt den Speicher mit konkretem Leben.
  Beispiele:

```
Person p;              // Deklaration der Variable p
p = new Person();      // Definition der Variable p

int jahrgang();        // Deklaration der Methode (so nicht in Java möglich)
int jahrgang() {       // Definition (Füllen der Signatur mit Code)
   // Methodenanweisungen
}
```

Die Unterscheidung der Begriffe *Objekt*, *Instanz*, *Attribut*, *Instanzvariable*, *Operation* und *Methode* hängt zudem von der Programmierumgebung (z.B. der Konvention in einer Programmiersprache) ab.
Andere Programmierumgebungen verwenden überdies auch noch weitere Begriffe wie z.B. *Object Type* für Klassen (OMG UML Modellierungsebene) oder *Member Function* für Methoden (Programmiersprache C++).

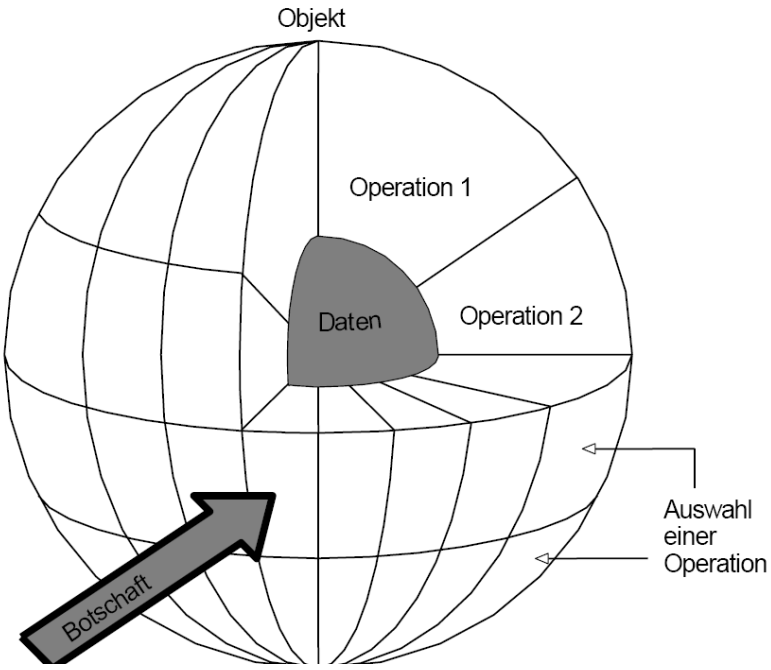
1.2 Modellierungsprinzipien

Umsetzung der wichtigen OO Konzepte bisher:

OO Konzept	Umsetzung
Beziehungen zwischen Klassen (und Objekten)	▪ Instanziierung: Beziehungen zwischen Objekten und Klassen durch die Instanziierung. ▪ Beziehungen zwischen Objekten durch Nachrichten und Methodenaufruf.
Abstraktion	Abstraktion vom konkreten Objekt durch Klassen
Kapselung / Geheimnisprinzip	Kapselung der Daten im Objekt
Modularität und Hierarchie	▪ Gruppierung von Daten und Operationen, die auf den Daten operieren (Abstrakter Datentyp, ADT) zu Modulen. Die Module heißen hier speziell Objekte und Klassen. ▪ Hierarchie in der Klasse-Objekt-Abstraktion.

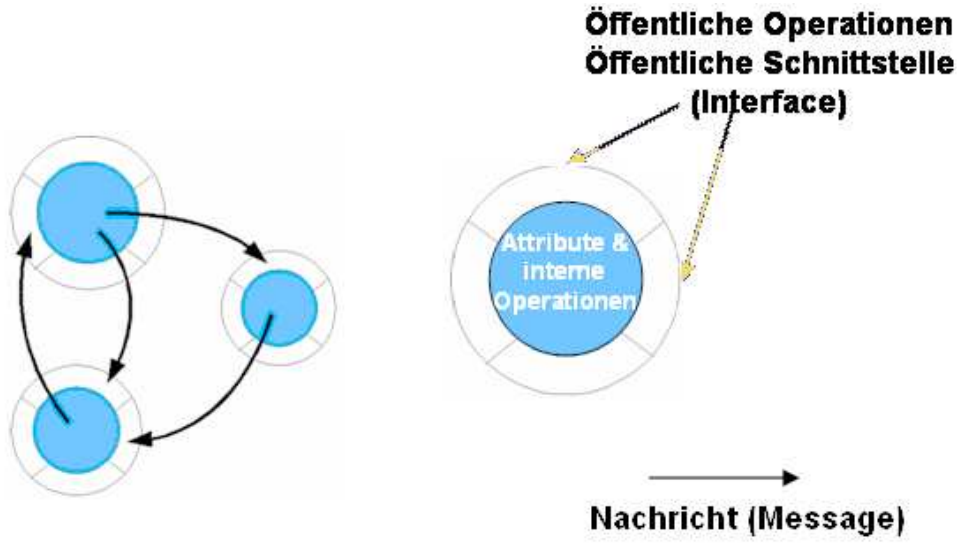
Kapselung und Geheimnisprinzip:

- Kapselung (engl. Encapsulation): Zusammenfassung von Operationen und Attributen zu einer Einheit (genauer Zusammenfassung von: Attributen, Operationen, Zusicherungen, Beziehungen; Zusicherungen und Beziehungen werden wir noch genauer kennen lernen)
- Geheimnisprinzip (engl. Information Hiding): bewusstes Verbergen von Implementierungsdetails (z.B. Attributinhalte); nach außen ist nur die Schnittstelle sichtbar



Im objektorientierten Paradigma werden Daten als ein zu schützender Schatz betrachtet. Von außen ist ein direkter Zugriff nicht möglich. Eine Botschaft erreicht das Objekt. Aufgrund dieser Botschaft wird eine Operation des Objekts ausgewählt und abgearbeitet. Gegebenenfalls greift diese Operation dann auf die internen Daten zu. Die Operationen des Objekts bilden einen Schutzwall um die internen Daten.

In begründeten Fällen kann von dieser Regel (dem Verbot eines direkten Zugriffs auf interne Daten von außen) auch abgewichen werden (solange man genau weiß, was man tut).



Warum nur gekapselter Zugriff?
Die Kapselung ergibt sich aus dem hohen Wert, der den Daten (z.B. Kundendaten in einem Unternehmen) zugemessen wird.
Durch den gefilterten Zugriff via Operationen kann bei späterer Programmänderung sehr schnell eine Anpassung realisiert werden.
Beispiel: In einer Programmänderung aufgrund neuer Datenschutzvorschriften sollen die Kundendaten nicht mehr mit vollständiger Telefonnummer, sondern nur noch mit den ersten zwei Ziffern der Rufnummer herausgegeben werden.
Direkt auf die Telefonnummer zugreifende Objekte müssen nun alle einzeln angepasst werden. Ist die Telefonnummer dagegen gekapselt und wird auf sie nur indirekt über eine Operation zugegriffen, genügt die Anpassung dieser einen Operation (Prinzip: Lokalisierung von Änderungen).

Entwurfsprinzipien:

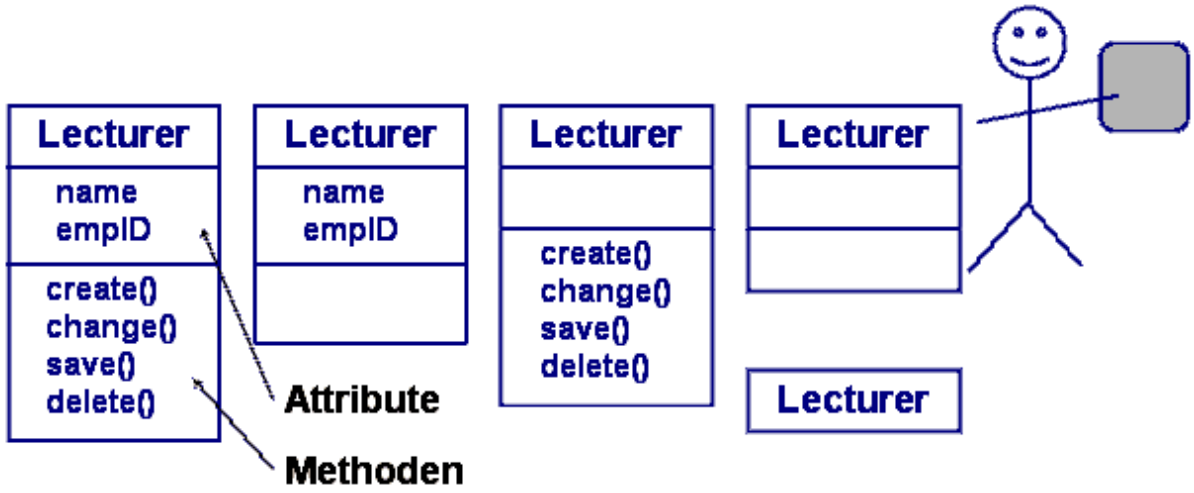
Mindestens genauso wichtig wie die objektorientierten Konzepte selbst, ist deren richtige Anwendung.
Dabei gibt es (leider) kein wirkliches „richtig“ oder „falsch“. Stattdessen gibt es nur „besser“ oder „schlechter“. Programmierung (und noch vielmehr die gesamte Entwicklung) ist keine streng mathematische oder mechanische Aktivität. Statt der Konzentration auf das letzte syntaktische und semantische Detail einer Sprache, sollten Sie daher vielmehr nach Erfahrungswerten und Richtlinien zur Anwendung suchen. Diese Erfahrungswerte und Richtlinien bilden den Grundstock, die Orientierung und das Futter Ihrer kreativen Entwicklungsarbeit.
Zentrale Entwurfsprinzipien, die man jenseits des Prinzips der Kapselung bei der Entwicklung eines objektorientierten Systems immer im Auge halten sollte, sind:

- Kohärenzprinzip bzw. Kohäsionsprinzip: Jede Klasse soll für genau einen (sachlogischen) Aspekt des Gesamtsystems verantwortlich sein.
- Objekt-Verantwortlichkeitsprinzip: Jedes Objekt – und nur dieses – ist für seine Eigenschaften selbst verantwortlich.
- Nachrichtenaustauschprinzip: Objekte sind eigenständige Einheiten, die durch Zusendung von Nachrichten miteinander interagieren.

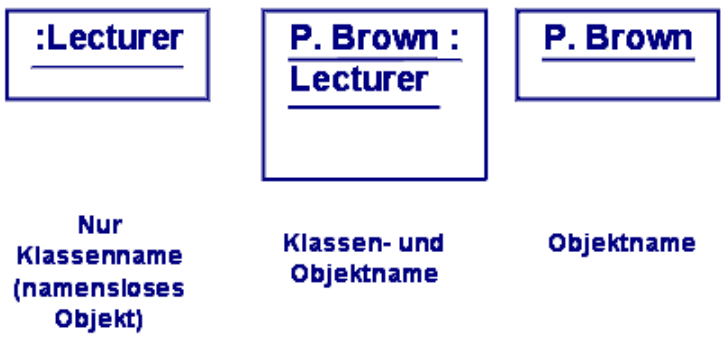
Vor allem das Kohärenzprinzip bzw. Kohäsionsprinzip und das Objekt-Verantwortlichkeitsprinzip spielen eine wichtige Rolle bei der Umgestaltung von Programmen (und Entwürfen). Jedes etwas größere Programm muss früher oder später während der Entwicklung neu gestaltet werden, um einen guten Aufbau zu erreichen bzw. zu erhalten. Es müssen z.B. neue Klassen oder Beziehungen eingefügt werden oder aber Klassen mit anderen verschmolzen werden. Diese Restrukturierung heißt auch Refaktorisierung (Refactoring).
Ein guter Aufbau ist wiederum z.B. die Grundlage für die Verständlichkeit und Weiterentwicklungsfähigkeit eines Programms.

Modellierungssprache: UML (Unified Modeling Language)

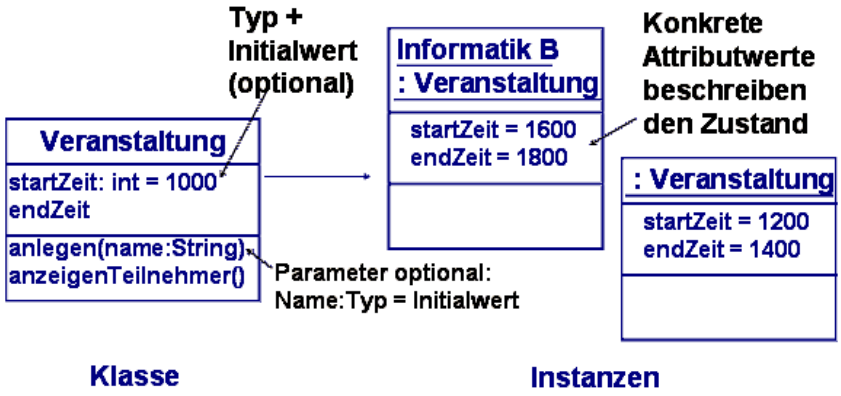
Mit Hilfe der Unified Modeling Language können die Konzepte eines Programms grafisch dargestellt werden.
Die Darstellung für Klassen sind Rechtecke, die aus folgenden Abteilungen bzw. Abteilen (engl. Compartments) bestehen: Klassenname, Attribute und Operationen.
Je nach Detaillierungswunsch können verschiedene Darstellungsalternativen unterschieden werden:



Darstellung von Instanzen in UML:
Wie Klassen werden auch Instanzen als Rechtecke gegebenenfalls mit Abteilen dargestellt. Die Darstellung unterscheidet sich in der Bezeichnung: Eine Instanz kann (wie im Beispiel im nachfolgenden Bild) ohne eigenen Instanznamen nur mit Typ (der Klasse), nur mit Namen aber ohne Typ oder sowohl mit Namen als auch Typ angegeben werden. In jedem Fall muss diese Bezeichnung unterstrichen werden. Das Modell eines Objekts kann immer nur ein Schnappschuss zu einem bestimmten Zeitpunkt sein. Die Attribute sind daher mit konkreten Werten belegt (obgleich diese in der Darstellung nicht zwingend angegeben sein müssen).



Beispiel für eine UML Darstellung einer Klasse und einiger Instanzen dieser Klasse (der Pfeil ist nicht Teil der Sprache UML):



Konkrete Wertangaben einer Klasse (z.B. für das Attribut `startZeit` in Klasse `Veranstaltung`) sind optionale Initialwerte. Bei Instanzerzeugung soll das entsprechende Attribut zunächst mit diesem Werte initial belegt werden. Analog kann auch bei den Methodenparametern ein initialer Parameterwert angegeben werden.
In der UML Darstellung der Instanzen sind die konkreten Werte dagegen tatsächliche Attributbelegungen zu einem bestimmten (wichtigen) Zeitpunkt im Verlauf des Systems. Sie stellen also einen einzelnen Zustand eines Objekts dar.
Das Beispiel zeigt darüber hinaus, wie in Klassen die Typen und Parameterlisten in UML notiert werden. Der Typ steht immer hinter dem Parameter- bzw. Variablennamen.
Die Attribute der Instanzen benötigen in der Regel keine Typangaben.

Bemerkungen zur Modellierung:

- UML hat viele Darstellungsalternativen. Insbesondere betrifft dies die Darstellungsgenauigkeit. Auf diese Weise kann sich der Modellierer auf das Wesentliche konzentrieren. Überflüssige und störende Details können je nach Bedarf weggelassen werden. Was dargestellt werden soll und in welcher Genauigkeit hängt also von der Modelliererin, dem Modellierungszweck (z.B. Dokumentation oder Entwicklung) und dem zu modellierenden Problem ab.
- Bei der Modellierung von Objekten in UML sollten zur besseren Lesbarkeit die Objekttypen (d.h. die Klassennamen) in der Regel angegeben werden.

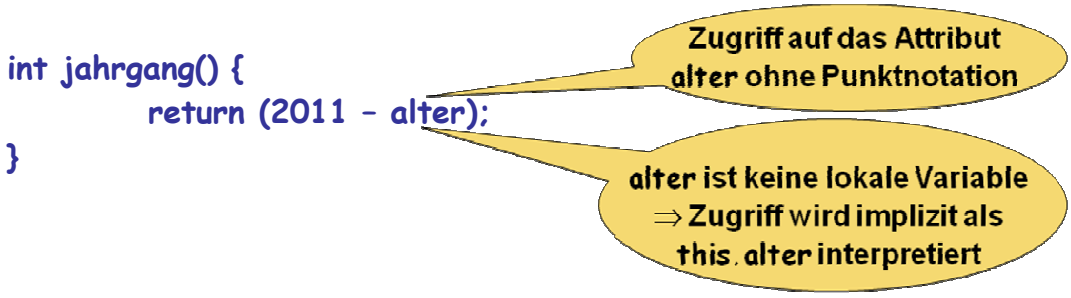
1.3 Identität: Zugriff auf sich selbst

In objektorientierten Programmiersprachen findet sich ein Schlüsselwort, das es Instanzen erlaubt, auf die eigene Identität zuzugreifen.
In Java heißt dieses Schlüsselwort (wie häufig auch in anderen Sprachen): `this`.

In Java gilt für dieses Schlüsselwort wie in den meisten OO Sprachen:

- `this` ist eine Referenzvariable, die immer auf die eigene Instanz verweist.
- `this` wird bei der Instanzerzeugung belegt ⇒ ist in der Instanz immer automatisch vorhanden.
- Mit `this` können eigene Attribute und Methoden angesprochen werden.
- Die Verwendung von `this` ist im Prinzip gleich wie für andere „normale“ Referenzen.
- `this` wird auch als versteckter Parameter an jede Operation übergeben (Ausnahme: Klassenoperationen – sehen wir später).

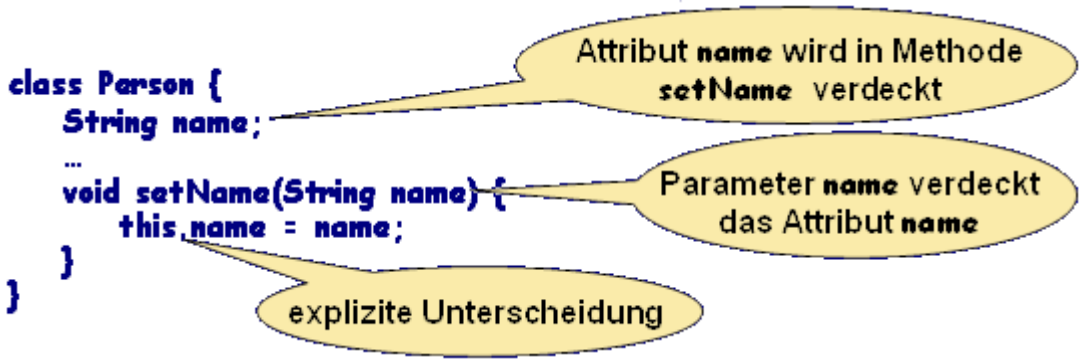
Hinweis zur Programmierung:
Greift eine Instanz auf die eigenen Objektelemente (Attribute und Methoden) zu, so kann dies explizit mit this geschehen oder ohne Angabe von this (this wird dann implizit ergänzt). Nicht selten ist ein expliziter Einsatz besserer Programmierstil, da es die Lesbarkeit verbessert.



1.4 Sichtbarkeitsbereiche (Scoping, Visibility)

Durch die Programmstruktur werden bestimmte Sichtbarkeitsbereiche festgelegt. Innerhalb von Methoden verdecken lokale Variablen und formale Parameter die Attribute gleichen Namens.

Mit Hilfe von this kann zwischen einer lokalen Variablen und einer Instanzvariablen (Attribut) gleichen Namens unterschieden werden.



Hinweis zur Programmierung:
Die Verwendung von gleichen Bezeichnern für verschiedene Elemente sollte vermieden werden, da dies die Lesbarkeit des Programms erschwert (z.B. nicht Bezeichner *name* für ein Attribut, eine Operation, einen Parameter und auch eine methodenlokale Variable im gleichen Programm). Die Mehrfachnutzung eines Bezeichners ist zudem eine vermeidbare Fehlerquelle.

Steuerung der Sichtbarkeit:
Neben den Sichtbarkeitsbereichen, die sich implizit aus der gebildeten Programmstruktur ergeben (z.B. Methodenvariablen, die Attribute verdecken), können Sichtbarkeitsbereiche auch explizit vom Programmierer festgelegt werden.
Methoden und Variablen können vom Entwickler für unterschiedliche Sichtbarkeitsbereiche definiert werden.
Dazu werden in Java die sogenannten Modifizierer bzw. Modifikatoren (engl. Modifier) eingesetzt.

Ein Attribut oder eine Methode, die als public deklariert wird, ist für dieselbe Klasse, andere

Klassen im selben Paket, Subklassen in einem anderen Paket und beliebige andere Klassen in anderen Paketen zugreifbar.
Die Zugriffsmöglichkeiten der übrigen Modifizierer (protected, private, ohne Modifizierer) sind in der folgenden Tabelle zusammengestellt.
Die erwähnten Konzepte „Paket“ und „Subklasse“ werden wir später noch genauer betrachten.

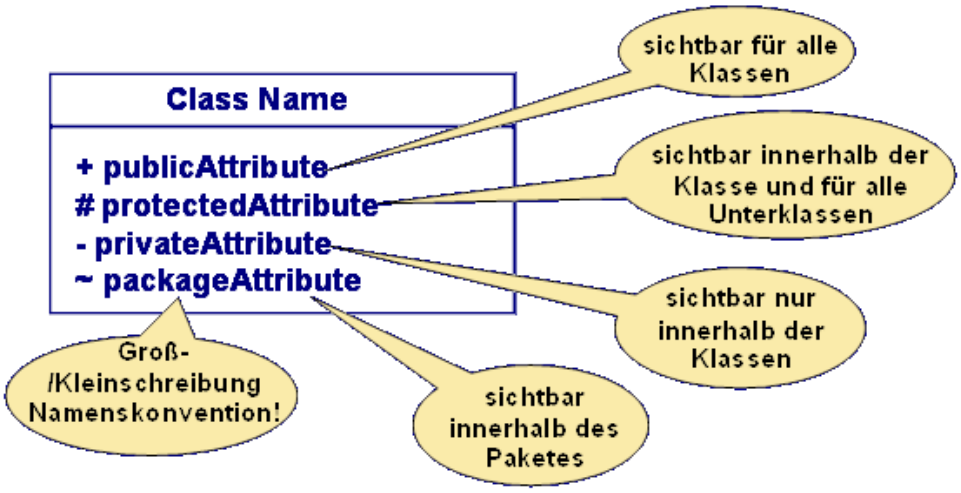
erreichbar für …	public	protected	paketsichtbar	private
dieselbe Klasse	Ja	Ja	Ja	Ja
andere Klasse im selben Paket	Ja	Ja	Ja	Nein
Subklasse in anderem Paket	Ja	Ja	Nein	Nein
keine Subklasse, anderes Paket	Ja	Nein	Nein	Nein

Bemerkung zur Programmgestaltung (Modellierung):
Das Geheimnisprinzip sollte Leitlinie der Entwicklung sein, d.h.

- Generell sollte man einen soweit wie möglich eingeschränkten Sichtbarkeitsbereich verwenden.
- Es sollte möglichst kaum „jemand“ auf Variablen direkt zugreifen können, es sei denn es ist zwingend erforderlich.

Hinweis zur Programmierung:
Die meisten objektorientierte Programmierumgebungen kennen die Sprachelemente public und private. Achtung: Die Sichtbarkeiten „paketsichtbar“ und „protected“ unterscheiden sich dabei unter Umständen. Darüberhinaus können in anderen Sprachen weitere Sichtbarkeitsregeln und Sprachelemente zur Steuerung der Sichtbarkeit enthalten sein (z.B. Schlüsselwort friend in C++, welches anderen „Freund“-Klassen den Zugriff auf private Elemente erlaubt!).

Darstellung von Sichtbarkeiten für Attribute in UML: (für Methoden analog)



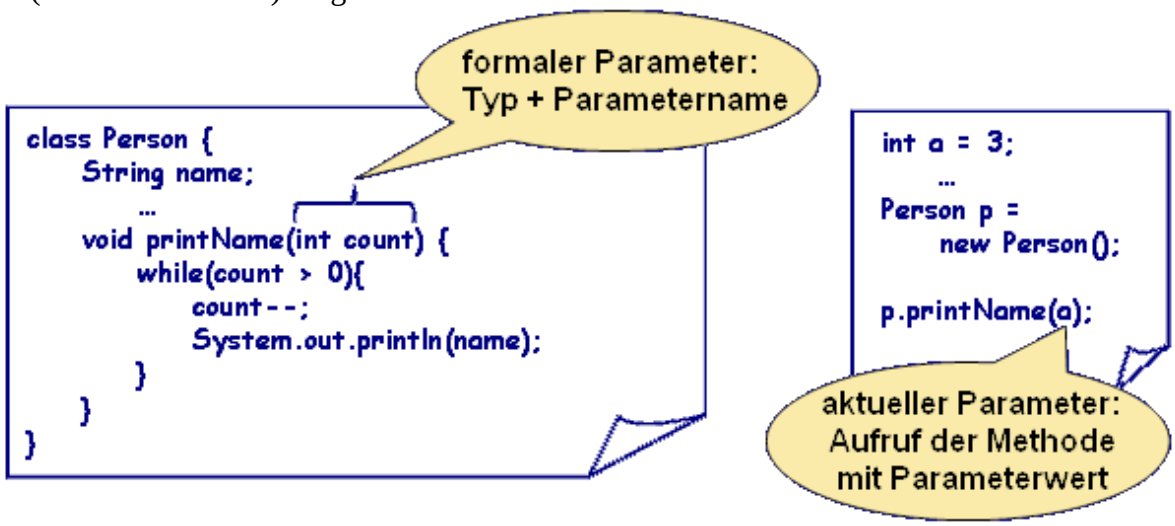
Achtung:
In der objektorientierten Welt von UML unterscheidet sich „protected“ semantisch zu Java: Eine andere Klasse, die keine Subklasse ist, aber im selben Paket liegt, kann in der Java-Welt

auf die „protected“-Elemente zugreifen. In UML geht das nicht (dort müssen auch die Klassen im selben Paket Subklassen sein, damit der Zugriff auf ein „protected“-Element möglich ist.)

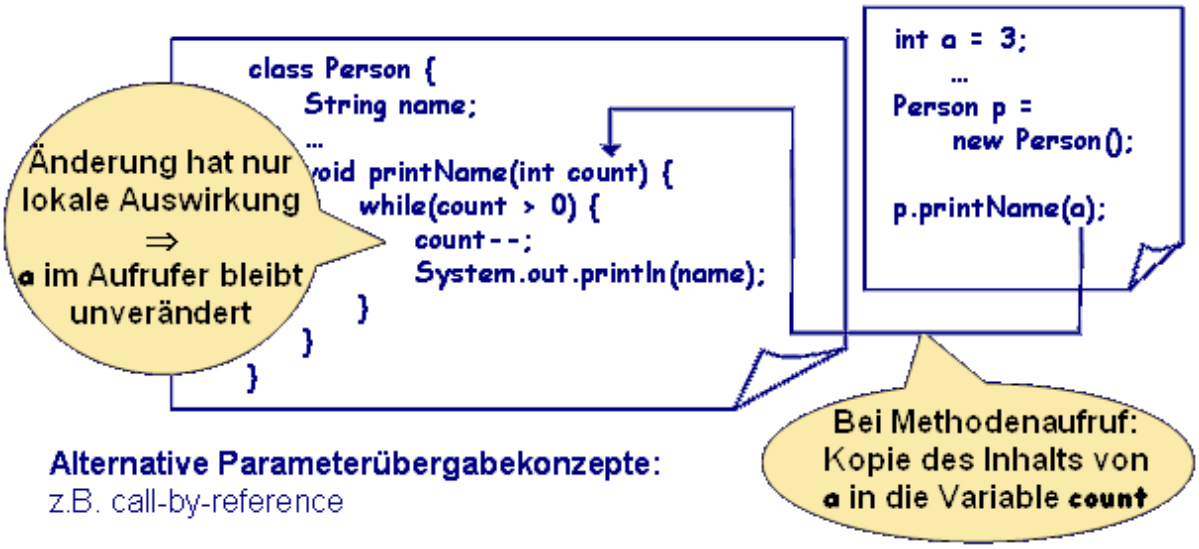
1.5 Methodenparameter und Rückgabewert

Methodenparameter und Rückgabewerte: Definition der Werte, die einer Methode beim Aufruf mitgegeben werden sollen bzw. die von der Methode zurückgeliefert werden.
In Java gilt (wie typisch in OO Sprachen):

- Definition der Parameterliste in Klammern hinter dem Methodenname.
- Soll mehr als ein Parameter definiert werden, so sind die einzelnen Definitionen durch Kommata zu trennen.
- Eine Methode wird mit formalen Parametern definiert und mit aktuellen Parametern (konkreten Werten) aufgerufen.



Parameterübergabekonzept:
Das Parameterübergabekonzept definiert nach welchen Regeln die Parameter übergeben werden.
Übergabekonzept für Parameter in Java: Call-by-Value



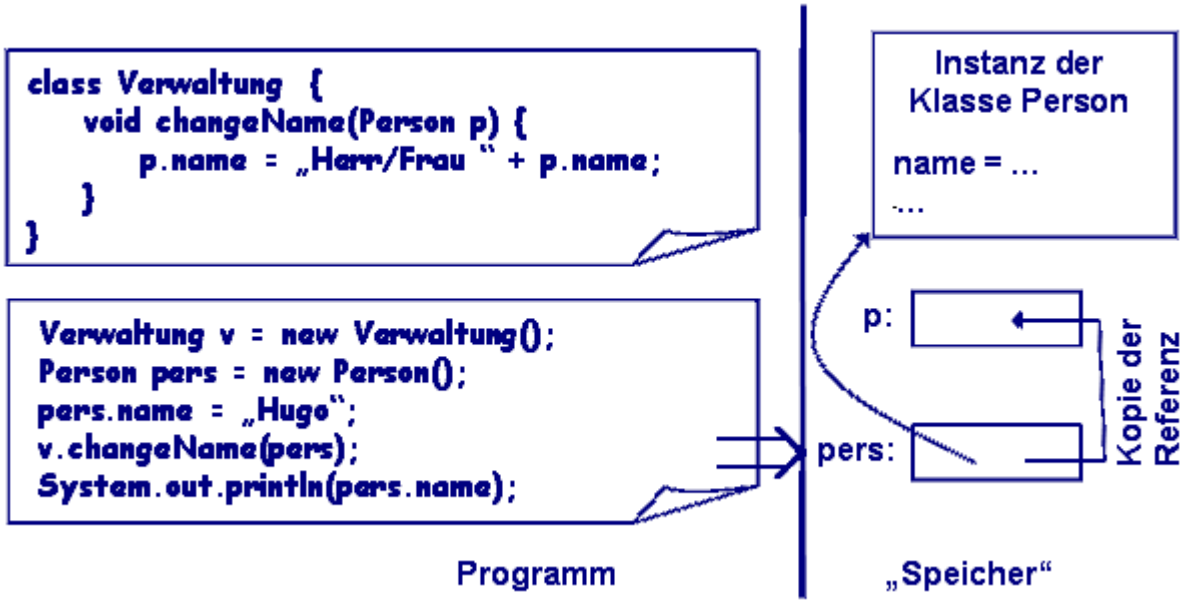
Andere Programmiersprachen unterstützen auch noch alternative Parameterübergabekonzepte, wie z.B. Call-by-strict-Value (keine Änderung am Parameter im aufgerufenen Objekt erlaubt) oder Call-by-Reference.
Beispiel: In C++ ist eine explizite Übergabe vom Typ Call-by-Reference möglich. Der übergebene Parameter wird als Speicheradresse (und nicht als unabhängige Kopie) übergeben. Änderungen am Parameter in der aufgerufenen Methode sind so auch im Aufrufer sichtbar.

Seiteneffekte treten trotz Call-by-Value auf:
Nicht primitive Datentypen (Referenzen) werden ebenfalls per Call-by-Value übergeben.
Da die Referenz kopiert wird, steht innerhalb der Methode ein Verweis auf das Originalobjekt zur Verfügung, d.h. der Aufgerufene arbeitet direkt auf dem Originalobjekt

- ⇒ Veränderung an diesem Objekt sind auch für den Aufrufer der Methode sichtbar (evtl. erwünschter Seiteneffekt)
- ⇒ Performante Übergabe, da nur eine Referenz übergeben wird (v.a. praktisch bei großen Objekten)

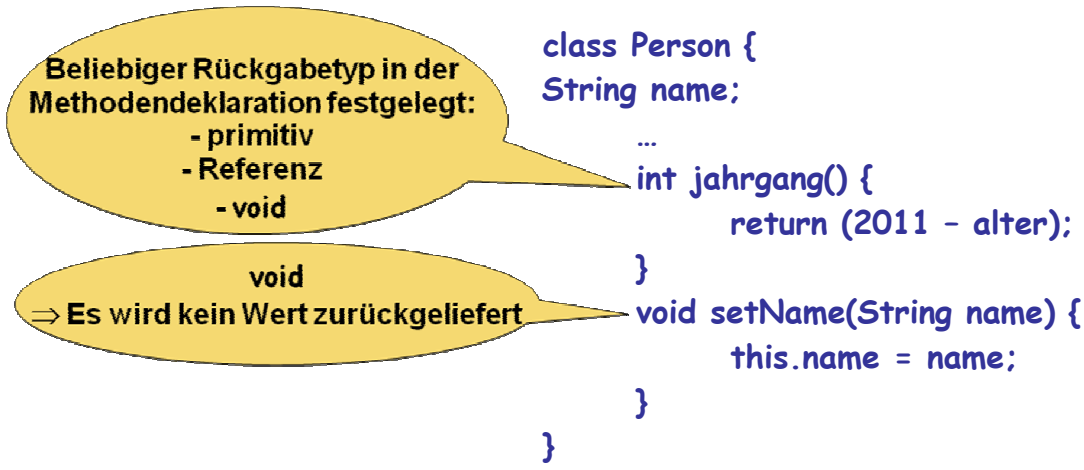
Vermeidung von Seiteneffekten: Durch explizite Kopien (z.B. mit der Methode `clone()`)

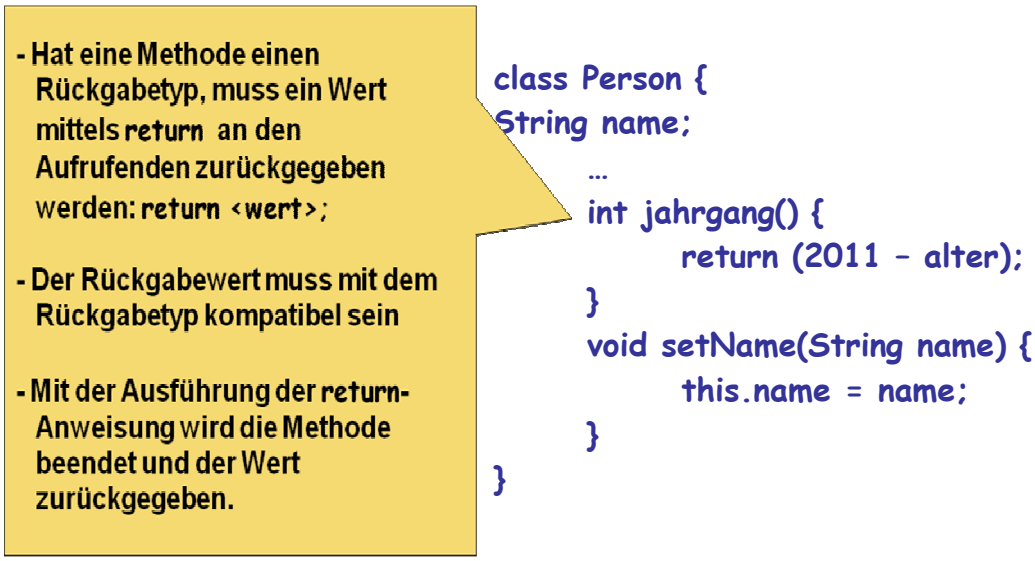
Beispiel für einen Seiteneffekt bei einer Call-by-Value Parameterübergabe:



Im Beispiel wird der aktuelle Parameter `pers` an den formalen Parameter `p` in Methode `changeName(Person p)` gebunden. Der Wert in `p` ist eine Objektreferenz. Dieser wird gemäß Call-by-Value in den Parameter `pers` kopiert. Im Bild ist rechts die Abbildung der Objekte im Speicher dargestellt (zumindest wie man es sich vorstellen kann; die Wirklichkeit des Speichers ist natürlich viel komplizierter). Nach der Kopieraktion zeigen `p` und `pers` auf das gleiche Objekt der Klasse `Person`. Änderungen an den Attributen des Objekts sind damit auch im Aufrufer sichtbar.
Seiteneffekte treten bei Referenzen als Parameter auf, nicht aber bei primitiven Typen.

Methodenrückgabewerte:



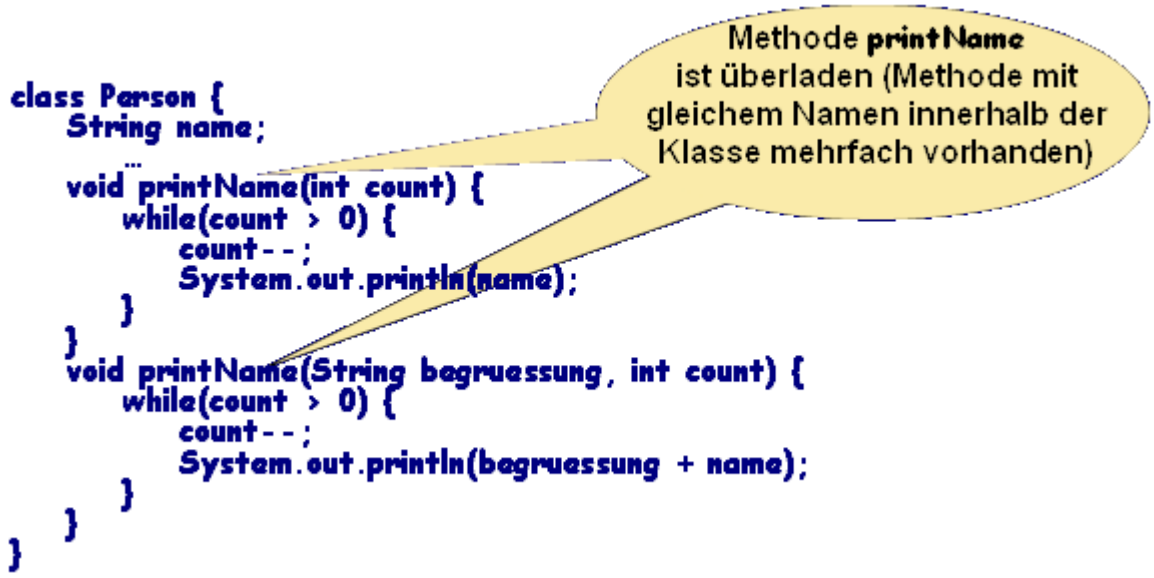


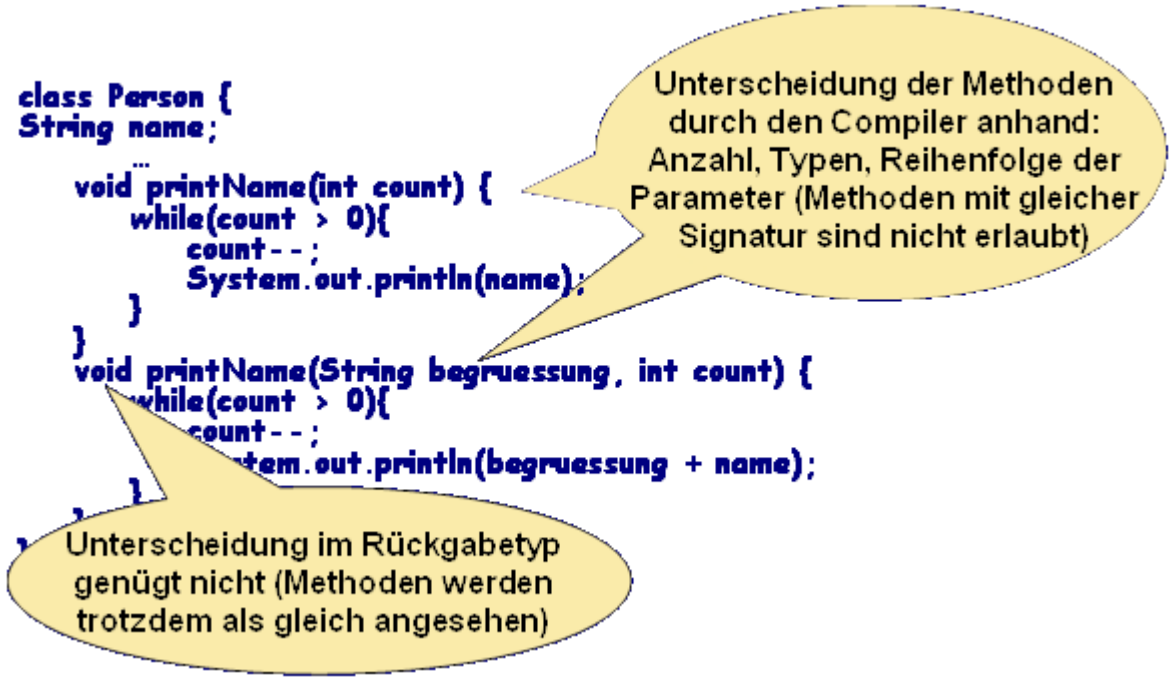
Wann wird void als Rückgabetyp verwendet?
Manche Methoden müssen nichts zurückliefern, sondern

- nur einen bestimmten Seiteneffekt erzielen bzw. interne Attribute ändern (Zustandsänderung)
- nur andere Methoden aufrufen bzw. externe Geräte ansteuern oder Ausgaben erzeugen.

1.6 Überladen von Methoden

Werden die gleichen Methodennamen (aber unterschiedliche Methodensignaturen!) in einer Klasse mehrfach verwendet spricht man von *Überladen*.
Beispiel in Java:





Spezialform des Überladens von Methoden: **Überladen von Operatoren** (engl. Operator overloading)

- Operatoren der Sprache (z.B. +, - oder Index-Operator []) werden in Abhängigkeit der Parametertypen mit neuer Bedeutung/Implementierung versehen.
- In Java gibt es nur typabhängiges Verhalten von Operatoren. Ein Überladen durch den Entwickler selbst ist nicht möglich. In C++ können Operatoren vom Entwickler überladen werden.
- Beispiel in Java: Überladener +-Operator (unterschiedliche Definition für Integer 1 + 2, String „abc“+“def“)

- Überladen kann den Code intuitiver machen, aber bei falschem Einsatz auch nur noch schwer nachvollziehbar.

Hinweise zur Programmierung:
- Das Überladen von Methoden ist nur sinnvoll, wenn gleichnamige Operationen auch eine vergleichbare, zum Methodennamen passende Funktionalität haben.
- Beim Überladen muss in einer Programmiersprache zunächst die Definition der Signatur beachtet werden (die gleiche Signatur darf nicht mehrfach in einer Klasse vorhanden sein, da sonst die Eindeutigkeit in der internen Behandlung von Methoden verloren geht).
  In Java ist der Rückgabetyp beispielsweise nicht Teil der Signatur. In anderen Sprachen muss das aber nicht gleichfalls gelten.

1.7 Spezielle Methoden

Arten von Methoden:

- new / delete (in Java: kein delete)
- Konstruktoren, Destruktoren
- Getter/Setter: Zugriffsmethoden (engl. Accessors) auf den internen Zustand
- sonstige Operationen (anwendungsspezifisch)

Diese Arten von Methoden finden sich in der Regel in allen OO Sprachen wieder.

new:

- Mit dem Operator new wird eine neue Instanz erzeugt.
- Für die Instanz benötigter Speicherplatz wird reserviert (z.B. für alle Attribute).
- Die Attributwerte werden mit Defaultwerten (Nullwerten) versehen.
  Einige wichtige Defaultwerte in Java:

int:	0
double:	0.0d
boolean:	false
Referenztypen:	null

- Eine Referenz auf das neue Objekt wird erzeugt und als Ergebnis von new zurückgeliefert.

Konstruktor (engl. Constructor):

- Spezielle Methode mit besonderen Eigenschaften und Aufgaben.
- Wird automatisch bei jedem new aufgerufen.
- Bei der Objekterzeugung kann der Aufruf eines Konstruktors nicht umgangen werden.
- Der Konstruktor versetzt ein mit new erzeugtes Objekt in einen sinnvollen Startzustand (Initialisierung und Prüfung der initialen Attributbelegung)
- Achtung: Der Konstruktor konstruiert kein Objekt, sondern „erzeugt“ sinnvolle Daten für das Objekt bzw. überprüft initiale Attributbelegungen.

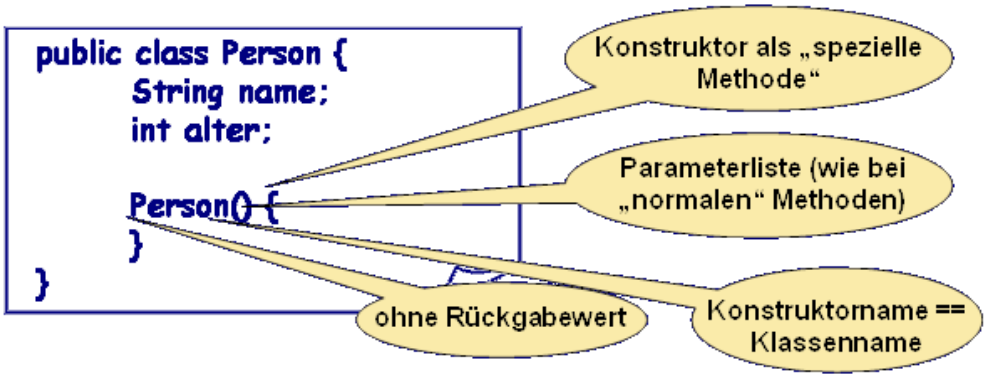
Allgemeine Java Syntax für Konstruktoren:

```
[Modifier] Klassenname([Parameterliste]) {
```

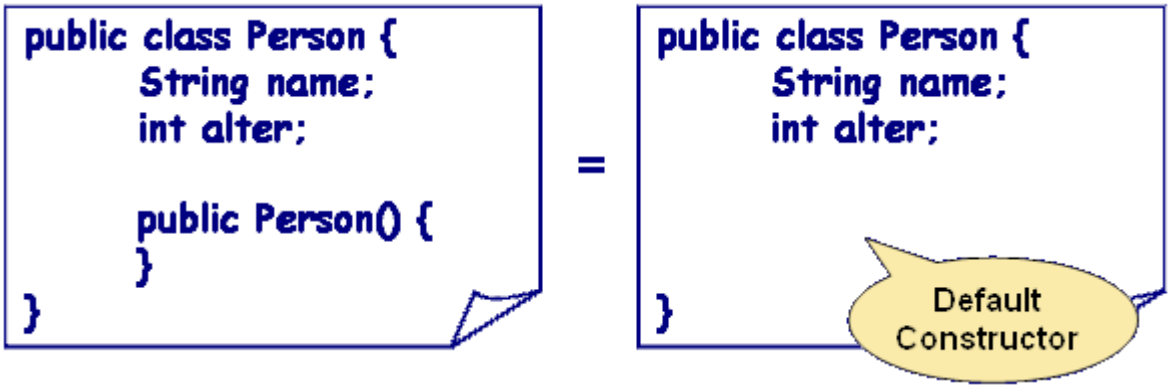


```
        // Anweisungen
}
```

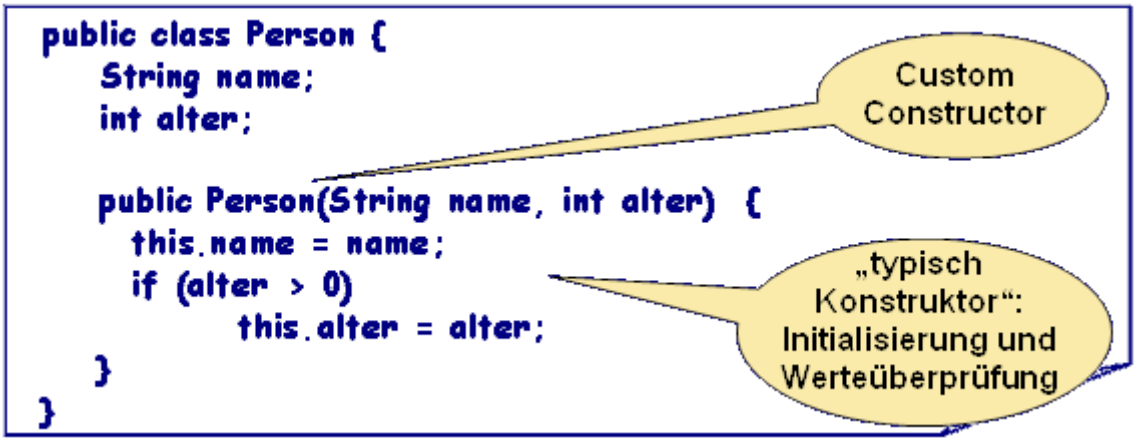
Beispiel in Java:



Default Constructor: Das ist der „Standard-Konstruktor“, der automatisch (nur) dann eingefügt wird, wenn kein Konstruktor explizit implementiert wurde (dafür sorgt in Java der Compiler). Jede Klasse hat damit mindestens einen Konstruktor.
Beispiel in Java: (im Bild rechts ist der implizite aber unsichtbare Default-Konstruktor, der sich wie der angegebene Konstruktor im Bild links verhält)



Custom Constructor: Ein benutzerdefinierter Konstruktor ist ein Konstruktor, der explizit vom Entwickler in einer Klasse ergänzt wird.
Beispiel in Java:



Konstruktion einer Instanz zum Beispiel: `Person aPerson = new Person(„Bart“,10);`

Achtung: In Java wird ein Default Constructor nur dann automatisch und implizit vom Compiler ergänzt, wenn kein anderer Konstruktor implementiert wurde.
Im Beispiel mit dem Konstruktor `Person(String name, int alter)` gibt es daher keinen Default Constructor.

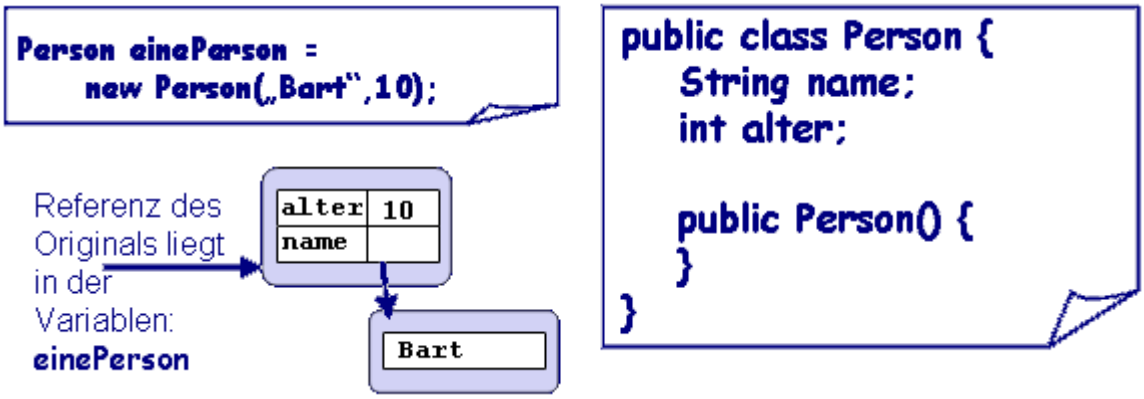
Hinweis für die Programmierung:
- Die Regeln für das automatische Einfügen von (Default-)Konstruktoren sind sprachspezifisch. In der Regel findet sich aber immer automatisch mindestens ein parameterloser Standardkonstruktor bzw. der Compiler erzwingt ggf. dessen Implementierung.
- Ob der explizit implementierte Konstruktor `public Person()` als Custom Constructor oder als Default Constructor bezeichnet wird, ist nicht eindeutig. Für beide Begriffe gibt es Gründe, die für und gegen die jeweilige Bezeichnung sprechen.

Überladen von Konstruktoren: Das Prinzip des Überladens von Methoden lässt sich auch auf Konstruktoren übertragen. Dem Nutzer einer Klasse werden mehrere Alternativen angeboten, wie Instanzen erzeugt werden sollen.
Beispiel mit zwei alternativen Konstruktoren in Java (Prinzip Überladen von Konstruktoren):

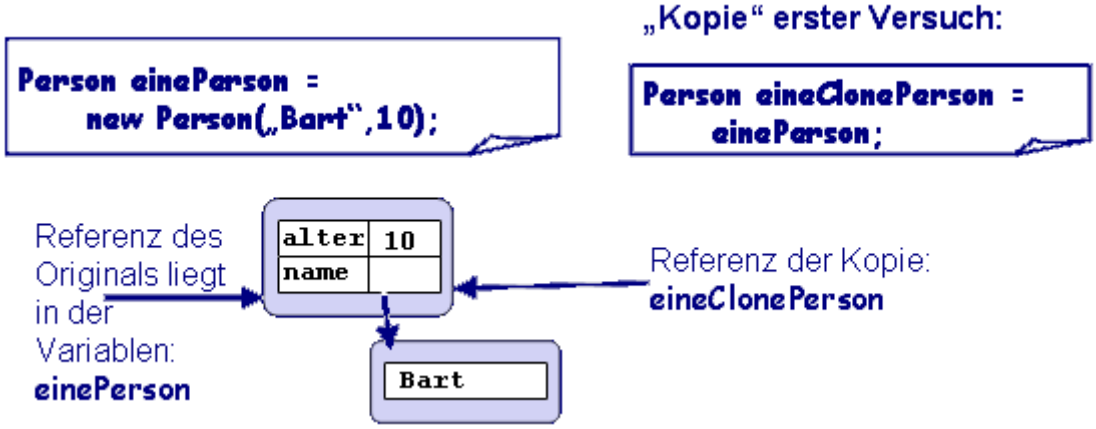
```
public class Person {
    String name;
    int alter;

    public Person() {
       name = „NA";
       alter = 0;
    }

    public Person(String name, int alter) {
      this.name = name;
      this.alter = alter;
    }
}
```

Der Compiler wählt den richtigen Konstruktor anhand der Parameterliste.

Der Copy Constructor:
Der Copy Constructor ist ein Konstruktor mit einer speziellen Aufgabe, dem Kopieren von Instanzen.
Aufgabe: Wir wollen eine Kopie eines bestehenden (originalen) Objekts erzeugen. Im Beispiel ist dies das Objekt `einePerson`.
Dazu legen wir das Objekt zunächst an und schauen, wie das Objekt im Speicher an einer bestimmten Speicheradresse (enthalten in der Variablen `einePerson`) mit seinen Attributen abgelegt ist. Das Attribut `alter` ist ein primitiver Datentyp, während der Name vom Typ `String` ist und daher im Attribut `name` selbst als Referenz auf ein separates String-Objekt abgelegt ist.

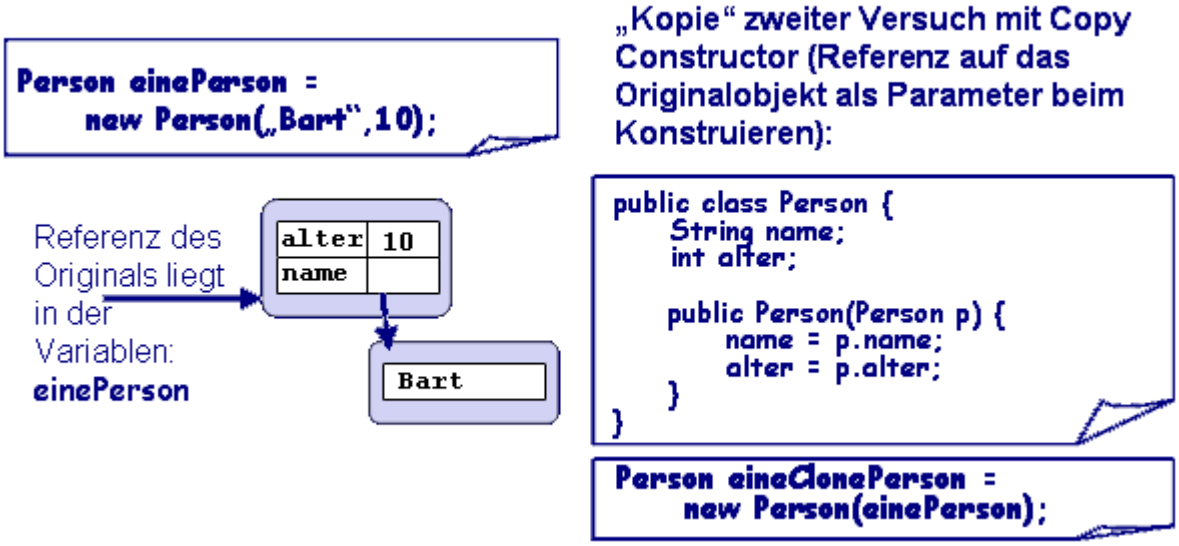


1) Erster Versuch einer Kopie: Kopie der Variableninhalte (**Kopie der Referenz**)

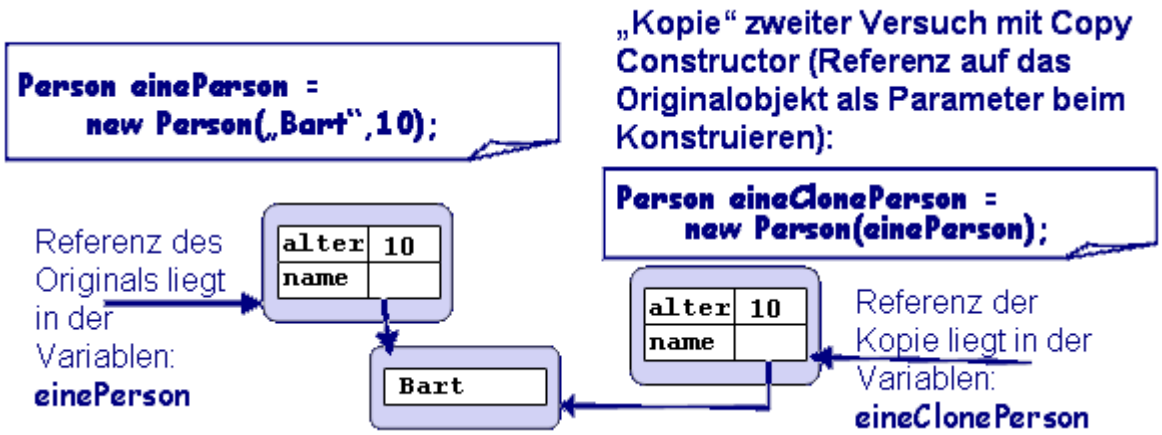


Problem: Die Kopie kann das Original ändern ⇒ keine „echte" Kopie (Seiteneffekte!)

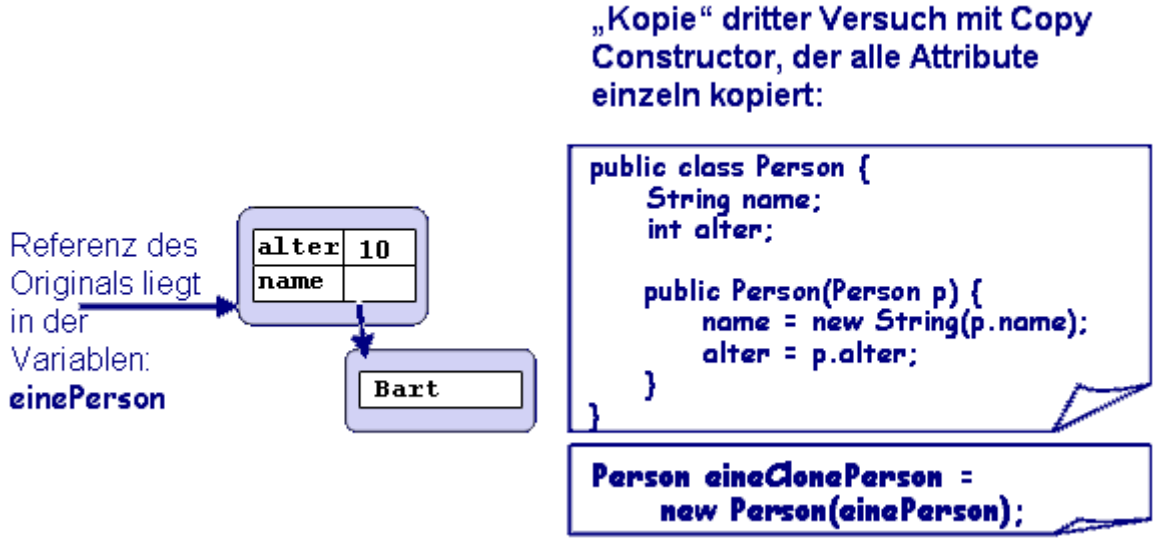
2) Zweiter Versuch einer Kopie: Copy Constructor (**Flache Kopie**, engl. **Shallow Copy**)

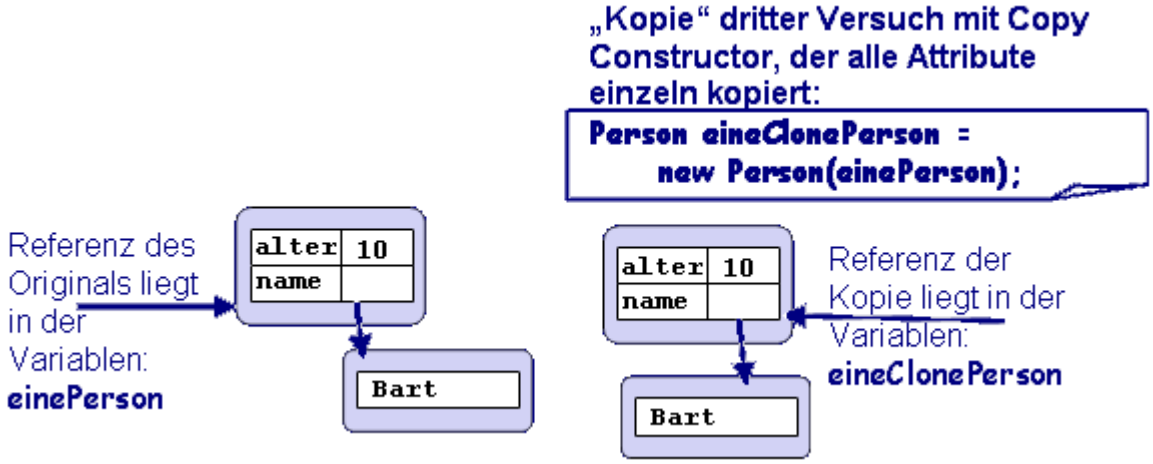


Problem: Die Kopie kann das Attribut name im Original ändern.
Begründung: Der Name ist ebenfalls ein Objekt und nicht nur ein primitiver Datentyp. Der Copy Constructor hat das Attribut name einfach als Referenz kopiert. In einePerson.name und eineClonePerson.name steht also die gleiche Speicheradresse (Lokation des Objekts, das den Namens-String enthält).



3) Dritter Versuch einer Kopie: Copy Constructor, der alle Attribute einzeln kopiert (**Tiefe Kopie**, engl. **Deep Copy**).
Primitive Datentypen können durch einfache Zuweisung kopiert werden. Für Objekte muss dagegen wieder eine neue Instanz erzeugt werden. Bei dieser neuen Erzeugung kommt wiederum der Copy Constructor zum Einsatz.





Gegenüberstellung der drei Kopieralternativen:
1) Kopie der Referenz (ohne Nutzung eines echten Copy Constructors)
⇒ gleiche Identität und damit keine „echte“ Kopie
2) Flache Kopie (Shallow Copy): Kopieren aller Attribute durch einfache Wertzuweisung (meist in der Initialisierung innerhalb des Konstruktors)
3) Tiefe Kopie (Deep Copy): Attribute, die Referenzen enthalten werden speziell kopiert („Verfolgen der Referenz“)

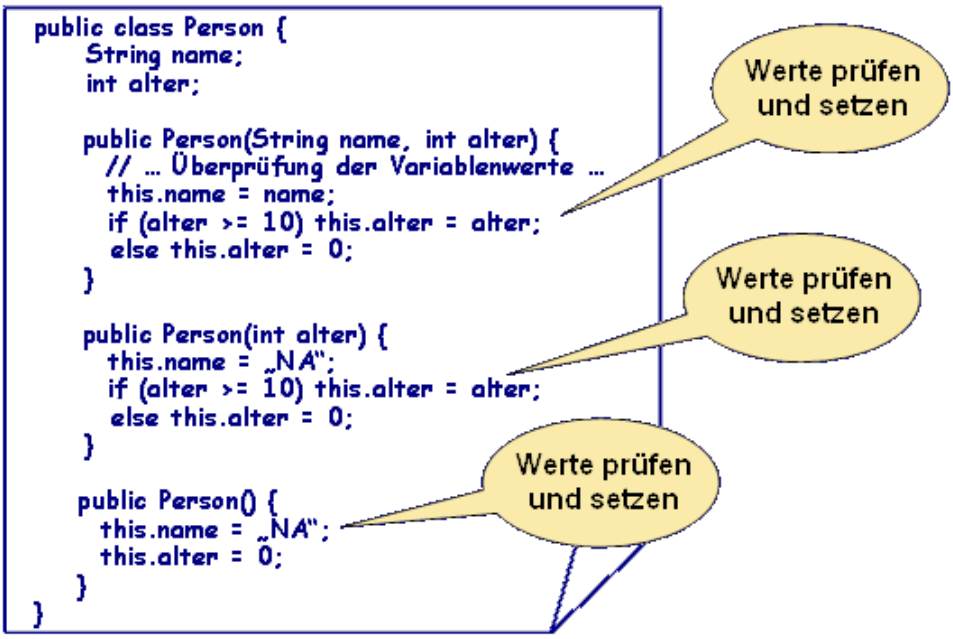
Achtung: Aufwand bei tiefen Objektverkettungen

Bemerkungen zum Deep Copy:
- Enthält eine Klasse nur primitive Datentypen, reicht eine Wertzuweisung bei den Attributen aus.
- Sind nur Referenzen auf nicht änderbare Objekte vorhanden, reicht ebenfalls eine Wertzuweisung aus.
  Eine echte Kopie ist - entgegen dem vorherigen Beispiel – beispielsweise bei `String`-Objekten nicht notwendig, da diese unveränderbar (engl. immutable) sind. Seiteneffekte können also sowieso nicht auftreten.
- Sind Referenzen auf veränderliche Objekte enthalten, müssen diese Objekte kopiert werden, wenn Seiteneffekte vermieden werden sollen.
- Im Zusammenhang mit Vererbung erweisen sich Copy Constructors als zu schwach (Abhilfe: `clone()` , Thema *Vererbung* folgt)

Hinweis zur Programmierung:
- Es kann vorkommen, dass Seiteneffekte erwünscht sind. Ein Programm mit solchen Seiteneffekten ist allerdings schnell nicht mehr zu überblicken. In der Regel sollten Objekte im Sinne der Nachvollziehbarkeit und Wartbarkeit gekapselt sein und nicht von außen (z.B. durch Seiteneffekte) direkt geändert werden können.
- Die beschriebenen Kopierprinzipien können in einem passenden Kontext auch in anderen Methoden (d.h. nicht nur in Konstruktoren) eingesetzt werden.
  In der Regel eignet sich für die Kopie der Copy Constructor des jeweiligen Objektes besonders.

Constructor Chaining (Verketten von Konstruktoren):

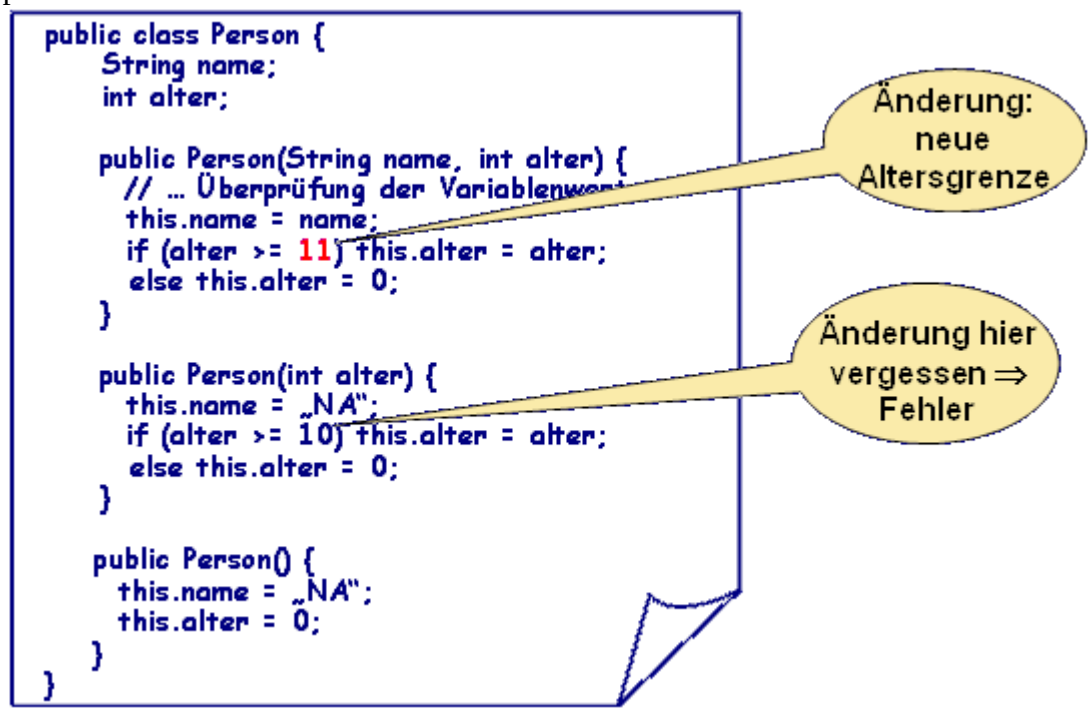
Problemdarstellung an einem Java Beispiel, welches Konstruktoren überlädt.



Problem: In den Konstruktoren werden jeweils die gleichen Aktionen (Setzen und Überprüfen) durchgeführt. Das Problem, dass gleiche Dinge mehrfach vorkommen, wird auch *Redundanz* genannt.
Nachteile von Redundanzen:

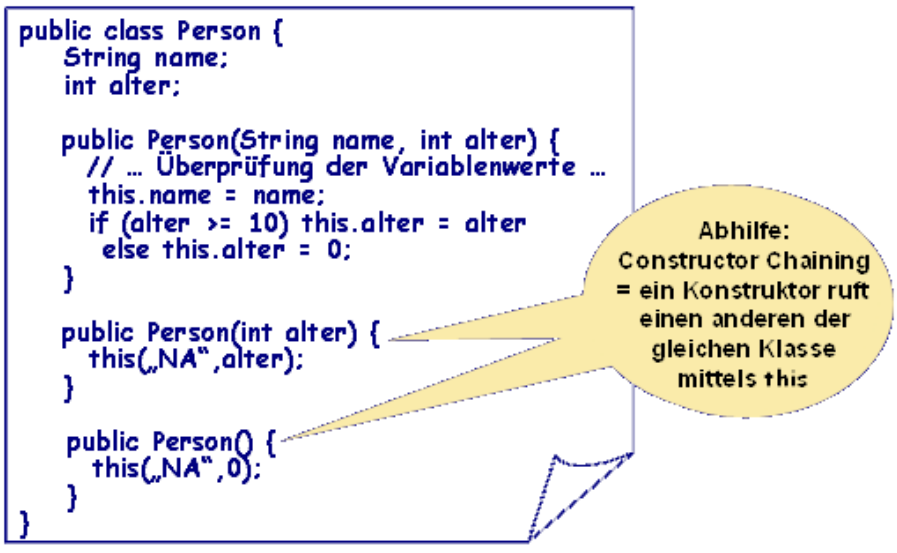
- mehr Code (Speicher!)
- mehr Aufwand (Codierung, Programmverstehen und Wartung)
- Fehleranfälligkeit, Inkonsistenz

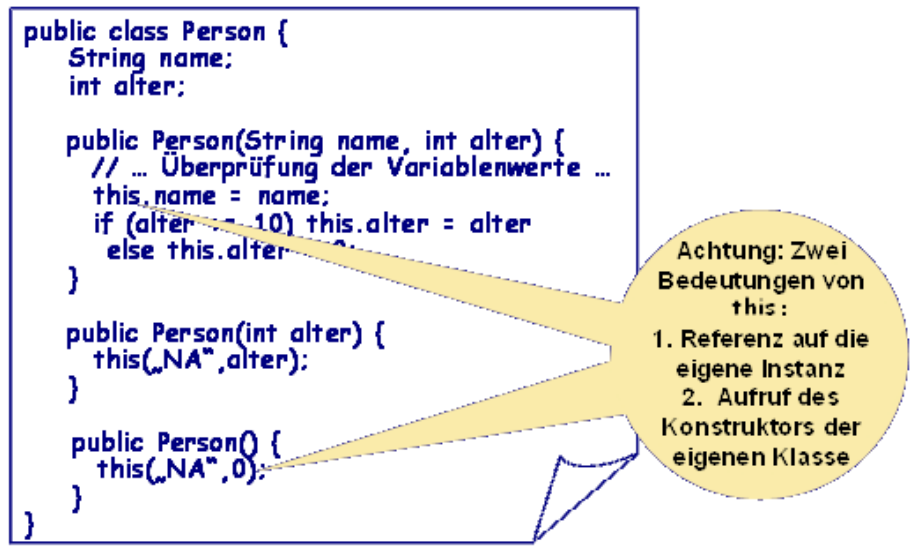
Beispiel:

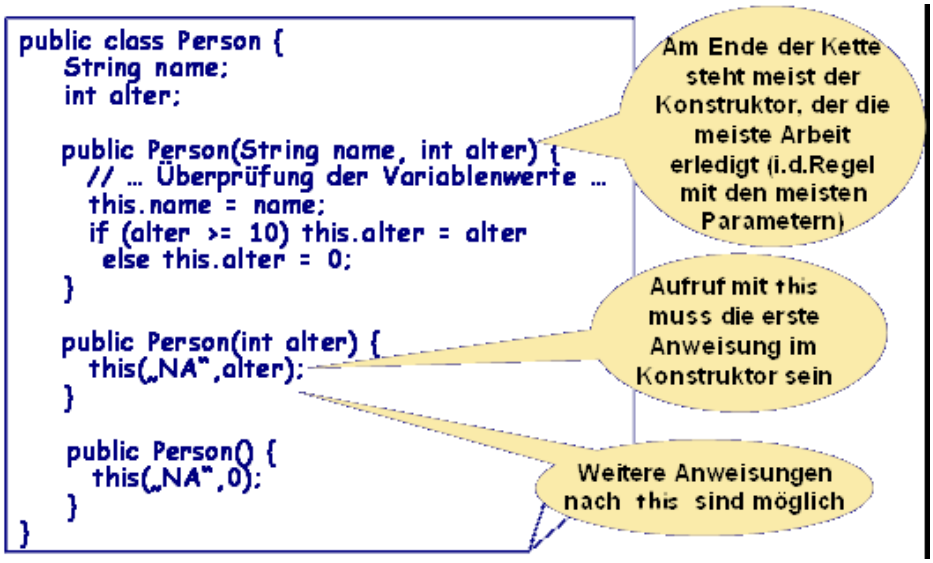


Abhilfe: Vermeidung der Redundanzen aus dem Beispiel durch Constructor Chaining

Constructor Chaining: Verketten von Konstruktoren, die sich gegenseitig aufrufen.
Beispiel:







Bemerkungen zum Constructor Chaining:

- Constructor Chaining ist nur sinnvoll, wenn die Konstruktoren ähnliche Aufgaben haben. Eine willkürliche Verkettung macht den Code nur unübersichtlich!

- Verkettungsrichtung: Am Ende der Kette steht in der Regel der Konstruktor, der die meiste Arbeit erledigt. Begründung: In umgekehrter Richtung würde ansonsten in jedem Konstruktor innerhalb der Kette ein Teil der Verarbeitung stattfinden. Für die Verständlichkeit des Codes gilt immer die Empfehlung, Aufgaben (hier die Initialisierung und Werteüberprüfung) an einem Ort zu lokalisieren (Lokalitätsprinzip) und nicht an viele verschiedene Orten zu verstreuen. Im Fall von Änderungen muss dadurch der Code z.B. nur an einer Stelle beachtet und angefasst werden.

- Eine Verkettung kann zwischen Konstruktoren innerhalb einer Klasse sowie über eine Klassenhierarchie (Vererbung) hinweg (oder auch beides) realisiert sein (Thema *Vererbung* folgt).

- Alternative: Implementierung einer speziellen Methode `initialize()`, die die Initialisierung übernimmt (Auslagerung des gemeinsamen Codes und jeweils Aufruf in den verschiedenen Konstruktoren).

Nachteile initialize()-Alternative:
(1) Der Compiler dupliziert letztlich den Code dann wieder (an den Aufrufstellen).
(2) Read-only Attribute können nur im Konstruktor direkt (einmalig) belegt werden.

Vorteile der initialize()-Alternative:
Diese Alternative lokalisiert die Aufgabe ebenfalls sehr gut an einer Stelle.
Eventuell ist diese Lösung sogar übersichtlicher hinsichtlich dessen, welche Parameter initialisiert werden. Beim Constructor Chaining ist eventuell nicht sofort ersichtlich, welcher Konstruktor die Arbeit letztlich tut.
Wie wir noch sehen werden, kann im Rahmen der Vererbung in einem Konstruktor der Konstruktor der Superklasse gerufen werden. Sowohl der Aufruf eines anderen Konstruktors der Klasse wie auch der Aufruf eines Konstruktors der Superklasse müssen allerdings die erste Anweisung in einem Konstruktor sein. Da beides nicht gleichzeitig machbar ist, bleibt oft nur der Weg, den Konstruktor der Superklasse zuerst zu rufen und die Initialisierungen der eigenen Klasse durch eine spezielle Initialisierungsmethode durchführen zu lassen.

- Das Prinzip der Verkettung von Konstruktoren bei ähnlichen, aufeinander aufbauenden Aufgaben, kann auch auf andere Methoden übertragen werden. Achtung: Auch hier ist bei der Verkettung auf die Übersichtlichkeit und auf das Lokalitätsprinzip zu achten.

Destruktor (engl. Destructor):

In OO Sprachen sind Destruktoren für die Aufräumarbeiten unmittelbar vor dem Zerstören des Objekts verantwortlich (z.B. Schließen von Netzwerkverbindungen und Dateien, Speicherfreigabe). Destruktoren sind im Allgemeinen sehr wichtig, in Java allerdings eher weniger. Der Grund dafür liegt im Garbage Collection und in der Java Spezifikation.

Exkursion „Müllsammlung in Java“: Garbage Collection

- Nicht referenzierte Objekte werden vom Garbage Collector (GC) der Java Virtual Machine (JVM) automatisch aus dem Speicher entfernt.
- Der Garbage Collector wird automatisch vom System aufgerufen oder kann mittels `System.gc();` gestartet werden.
- Unmittelbar vor dem Zerstören eines Objektes durch den GC wird ein Destruktor (Destructor) aufgerufen:

  In Java heißt der Destruktor auch Finalizer:

```
protected void finalize() {
    // Anweisungen
}
```

⇒ Java hat eine automatische Speicherverwaltung (den Garbage Collector) hinsichtlich der Speicheraufräumarbeiten; Destruktoren sind daher weniger wichtig als z.B. in C++.

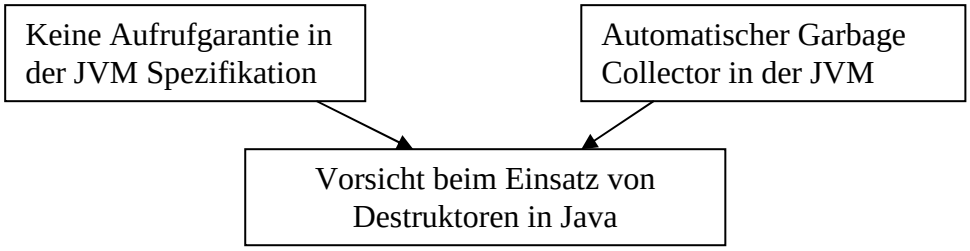
Hinweis zur Programmierung:
Wenn Objekte nicht mehr benötigt werden, empfiehlt es sich gelegentlich die entsprechenden Variablen in Java explizit auf null zu setzen. Damit kann sichergestellt werden, dass die Referenz wirklich „gelöscht“ ist und dass der Garbage Collector die Objekte bei nächster Gelegenheit abräumt und kein Speicher unnötig belegt bleibt (Achtung: in anderen

Programmiersprachen ohne Garbage Collector wird durch ein solches null-Setzen der Speicher nicht automatisch freigegeben). Besser noch als null-Setzen ist es, für Variablen einen möglichst engen Scope (Sichtbarkeitsbereich) zu wählen. Sobald der Scope verlassen wird, wird die Variable mit Inhalt aufgelöst. Eine Variable (mit enthaltener Referenz), die in einem if-Block definiert wurde, wird beispielsweise mit Verlassen des if-Blocks wieder aufgelöst. Das Objekt, auf das die Referenz in der Variable zeigte wird bei nächster Gelegenheit abgeräumt, sofern die Referenz nicht aus dem if -Block „herausgeschmuggelt" wurde, so dass sie nach Verlassen des if –Blocks noch an einer anderen Stelle zugreifbar ist.

Exkursion Java Virtual Machine (JVM) Spezifikation:
Um unabhängig zu einer konkrete Implementierung zu bleiben, definiert die JVM Spezifikation keine Implementierungsdetails, d.h. z.B. keine Spezifikation der genauen Implementierung des Garbage Collectors der Java Virtual Machine (JVM).

⇒ Aufrufzeitpunkt des Garbage Collectors und damit der Destruktoren ist nicht definiert, Aufrufreihenfolge der Destruktoren ist nicht definiert;
Aufruf ist nicht garantiert (wird das Programm z.B. vor Aufruf des GC beendet, wird der Destruktor nicht aufgerufen)
Auch der Einsatz von System.gc(); liefert keine Garantie.

⇒ Einsatz von Destruktoren in Java mit Vorsicht (allgemein sonst aber wichtig!)



Bemerkung zum Java Destruktor:
- Durch das gegenüber den Destruktoren anderer Programmiersprachen sehr unzuverlässige Verhalten von finalize() in Java wird teilweise (sehr sinnvoll) zwischen den Begriffen Destructor und Finalizer unterschieden.
- Durch das unzuverlässige Verhalten des Finalizers in Java wird teils ganz von deren Verwendung abgeraten. Statt dessen sollten Aufräumarbeiten in try-finally-Blöcken realisiert werden (Thema folgt später) oder in Terminierungsmethoden, die explizit aufgerufen werden müssen und die nach deren Ausführung die Instanz am besten explizit als ungültig markieren.

1.8 Gleichheit bzw. Identität von Objekten

Identität und Gleichheit: „Zwei Objekte sind identisch, wenn ihre Identität gleich ist."
Wir haben oben bereits gesehen, dass zwei verschiedene Objekte grundsätzlich zwei verschiedene Identitäten haben, dass es aber gleichzeitig schwierig ist, diese Identität mit einer Programmiersprache zu fassen.

In den meisten Fällen wird die Speicheradresse eines Objekts als Identitätsrepräsentation verwendet. Da in einem Speicher nur ein Objekt an einer bestimmten Adresse liegen kann, scheint diese Abbildung angemessen. In einer verteilten Umgebung, in der mehrere Rechner (mit jeweils ihren eigenen Speicherräumen) interagieren und Objekte zwischen den Rechnern wandern, ist die Identität von Objekten nicht mehr unmittelbar mit deren Speicheradressen gleichsetzbar.

Zur Unterscheidung zwischen Gleichheit und Identität können wir uns eineiige Zwillinge vorstellen. Die Zwillinge sind gleich aber nicht identisch.
Wann wir zwei Personen als gleich bezeichnen, können wir selbst je nach Vergleichziel festlegen. Beispielsweise können wir zwei Personen mit braunen Haaren bereits gleich nennen (das Gleichheitskriterium „Haarfarbe“ ist bei beiden mit dem gleichen Wert belegt). Identisch sind die beiden Personen deshalb trotzdem nicht. Umgekehrt können wir verlangen, dass zwei Objekte erst dann als gleich bezeichnet werden sollen, wenn sie identisch sind (d.h. also tatsächlich nicht zwei, sondern nur ein Objekt sind).

Die Gleichheit bzw. Identität kann je nach Anwendungsbereich daher jeweils auf verschiedene Weisen definiert werden:

- Namensgleichheit: Meist keine gute Idee zur Repräsentation der Identität
  Sind eindeutige Namen möglich und verwendbar? Ist die Eindeutigkeit garantiert? Ist der Namen unveränderbar?
  Beispiel für eindeutige Namen: Sozialversicherungsnummer (evtl. allerdings aus Datenschutzgründen nicht verwendbar)
- Künstlicher, unveränderbarer Schlüssel (z.B. Zeitstempel, Zähler, Netzwerkkartennummer)
- Gleichheit der Attribut(werte): Meist keine gute Idee, wenn die Identität damit abgebildet werden soll.
- Adressierbarkeit: Physikalisch identisch durch eine gleiche Speicheradresse (in Java standardmässig mittels Operator `==` testbar).

Bei Objekten muss jeweils aus der Problemdomäne (der Fachlichkeit) heraus festgelegt werden, wann zwei Objekte als gleich bezeichnet werden sollen sowie durch welche Mechanismen die Identität im Programm abgebildet werden kann (Entwurfsentscheidung).
In Java wird Gleichheit durch Überschreiben der Methode `equals(Object o)` festgelegt.

Bemerkungen:

- Methode `equals(Object o)` wird von `java.lang.Object` an alle Objekte vererbt und muss überschrieben werden, wenn die Gleichheit nicht durch die Identität definiert sein soll (Thema *Vererbung* und *Überschreiben* kommt noch).
- Die Methode `hashCode()` liefert für zwei gleiche Objekte (nach Definition von `equals()`) denselben Wert zurück
  In der Regel muss also `hashCode()` überschrieben werden, wenn `equals()` überschrieben wird.
- `Comparable` und `Comparator` sind Konzepte, die im Java Collection Framework zum Vergleich von Objekten enthalten sind und die ebenfalls für die Feststellung von Gleichheit eingesetzt werden können. Das Thema Java Collection Framework behandeln wir später.

Überschreiben von `equals()` erfordert die Einhaltung folgender Regeln:

- Symmetrie: Für zwei Referenzen x und y ungleich null gilt: x.equals(y) genau dann wenn y.equals(x).
- Reflexivität: Für die Referenz x ungleich null gilt: x.equals(x) ist true.
- Transitivität: Für drei Referenzen x, y und z ungleich null gilt: Wenn x.equals(y) und y.equals(z) dann auch x.equals(z).
- Konsistenz: Für zwei Referenzen x und y ungleich null liefert der wiederholte Aufruf von x.equals(y) entweder immer true oder immer false, falls sich keine Informationen ändern, die zur Berechnung von equals() verwendet werden.
- Für jede Referenz x ungleich null gilt: x.equals(null) liefert false.

1.9 Klasseneigenschaften

- In Java gibt es keine globalen Funktionen oder globale Variablen, die von überall zugreifbar sind.
- Manchmal möchte man Variablen oder Operationen haben, die nicht an Objekte einer Klasse gebunden sind:
  a) Klassenvariablen oder –attribute (engl. Class Scope Attribute):
  - existiert pro Klasse einmal
  - alle Objekte "teilen" sich die Variable
  - Zugriff in Java: Klassenname.Variablenname

  b) Klassenmethoden:
  - Zugriff in Java: Klassenname.Methodenname
- Deklaration von Klasseneigenschaften in Java mit Hilfe des Modifikators static

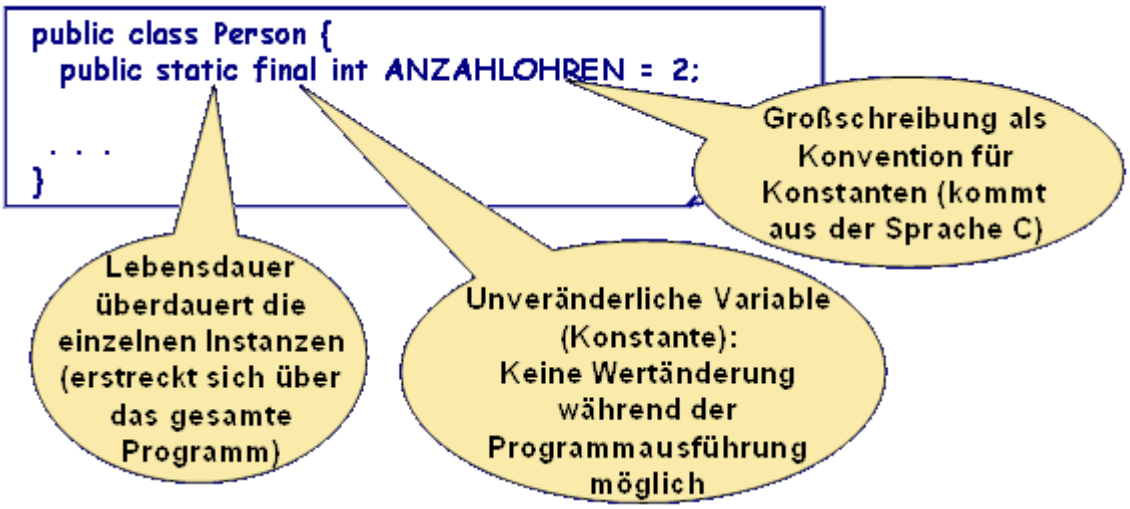
Hinweis zur Programmierung:
Klasseneigenschaften sind ein allgemeines objektorientiertes Konzept. Es findet sich nicht nur in Java, sondern in allen objektorientierten Sprachen mit in der Regel gleichem Verhalten. Unterschiede kann es allerdings in der Syntax und beim Klasseninitialisierer (vgl. unten) geben.

Beispiel Klassenattribut: Zählen der von einer Klasse erzeugten Instanzen.
Der Inhalt der Variablen objNo ist für alle Instanzen der Klasse Person gleich und sichtbar.

```
public class Person {
  static int objNo = 0;
  public String name;

  Person() {
    name = „dummy";
    ++objNo;
  }
  void dying()
  {
    --objNo;
  }
}
```

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

   Person p1 = new Person();
   Person p2 = new Person();
   System.out.println(„No: "
         + Person.objNo);
   p2.dying();
   System.out.println(„No: "
         + Person.objNo);

}
```

Beispiel Klassenattribut: Deklaration von Konstanten für alle Instanzen der Klasse:



Beispiel Klassenoperation: Verwendung von sqrt in der Klasse Math:

```
public static void main (String[] args)
{
   double x =4.0d;

   System.out.println(„Quadratwurzel ist: " + Math.sqrt(x));
}
```

Klasseneigenschaften finden sich z.B. in Klassenbibliotheken.
Beispiel für Klasseneigenschaften in Java:

- Klasse System: Diese Klasse ist eine Art "Toolbox" und dient insbesondere zur Kommunikation mit dem Betriebssystem. Sie enthält z.B. Funktionen für den Aufruf des Garbage Collectors oder für das Beenden des Programms. Ein bekanntes Beispiel ist die Klassenvariable out, die in System.out verwendet wird, um eine Ausgabe auf dem Bildschirm (über das Betriebssystem) zu realisieren.
- Klasse Math (mit Funktionen zur Fließkommaarithmetik als Klassenoperation, Hilfsklasse)

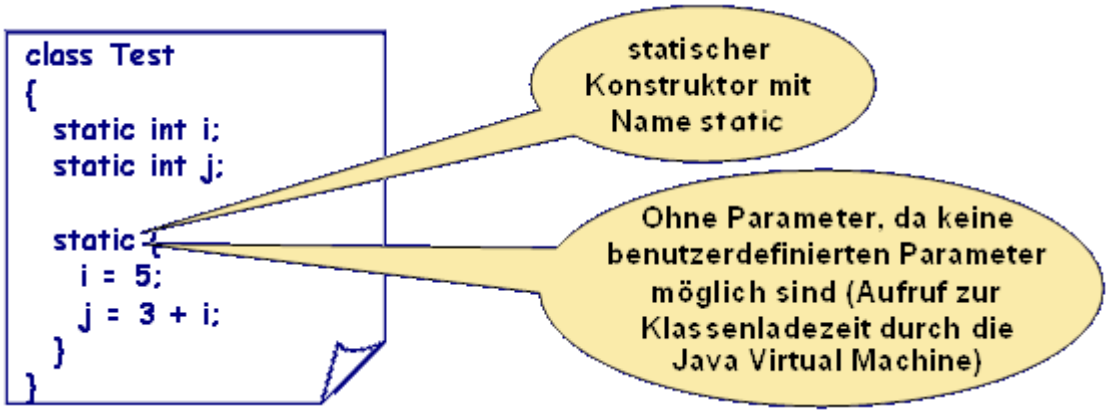
Besonderheiten von Klassenmethoden:

- kein Zugriff auf Instanzvariablen (Unabhängigkeit zum konkreten Objekt)
- this-Referenz unbekannt
- beides wird bei der Übersetzung vom Compiler als Fehler erkannt

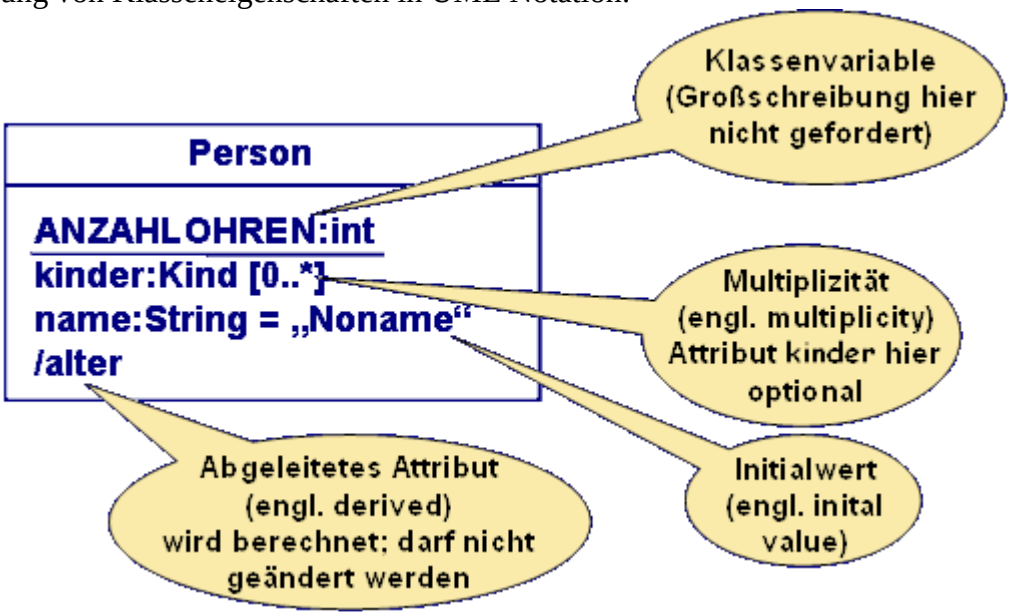
Statische „Konstruktoren" (Klasseninitialisierer)

- Sie sind ähnlich wie die schon bekannten Konstruktoren, dienen ihnen gegenüber aber der Initialisierung von Klassenattributen.
- Der statische Konstruktor wird einmal zu Programmbeginn (zum Zeitpunkt, zu dem die Klassen in die Java Virtual Machine geladen werden) aufgerufen (statt bei Instanziierungen wie bei den „herkömmlichen" Konstruktoren).

In Java:



Darstellung von Klasseneigenschaften in UML Notation:



Die UML Darstellung für Klassenmethoden ist analog zu der für Klassenattribute (mit Unterstreichung).
Das UML-Beispiel oben führt darüber hinaus zusätzliche Notation ein (Multiplizität, abgeleitete Attribute).
Im Beispiel kann eine bestimmte Person entweder keine (0) oder beliebig viele Kinder haben (*). Die Referenzen auf die Kindinstanzen werden im entsprechenden Feld gespeichert.
Beispiel für ein abgeleitetes Attribut: Das Alter einer Person ist ein abgeleitetes Attribut, wenn es nicht explizit als Wert gespeichert, sondern berechnet wird (z.B. aus dem gespeicherten Geburtsdatum und dem aktuellen Jahr).

Bemerkung zur Modellierung:
Am abgeleiteten Attribut kann man erneut den Unterschied zwischen Modellierung und Programmebene erkennen. Im Modell werden abgeleitete Attribute genannt, falls sie wichtig sind. In der Implementierung sind sie aber in keinem Fall (direkt) zu finden.
Ein Modell hat also in der Regel keine 1:1 Abbildung in eine Implementierung.

1.10 Der Singleton

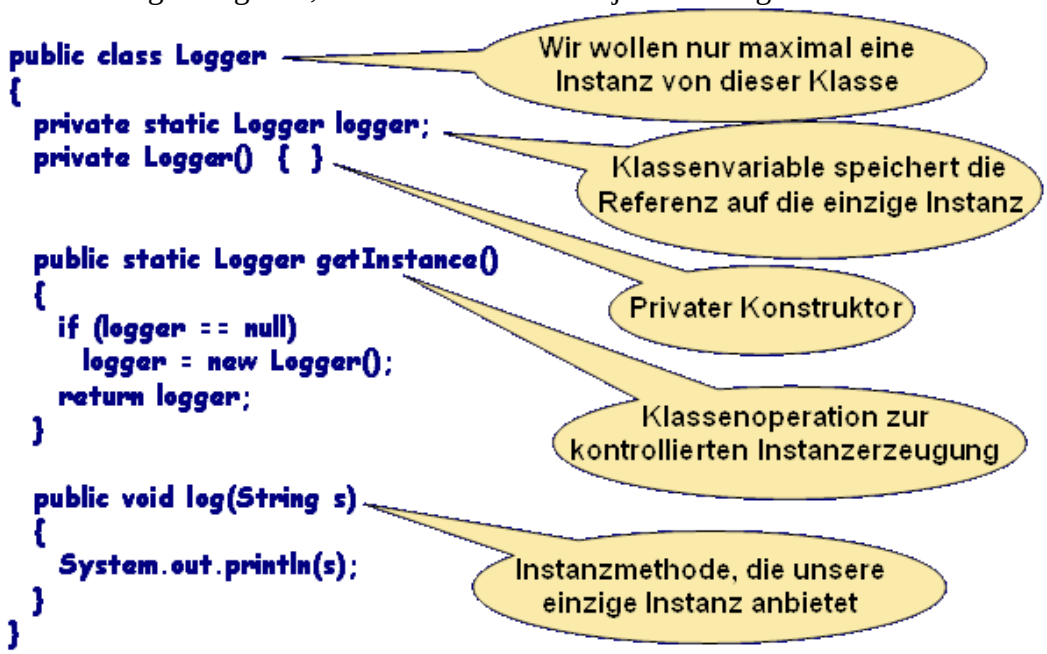
Ein typisches Einsatzszenario für Klasseneigenschaften ist das Entwurfskonzept des sogenannten Singleton (Einzelstück).

Problemstellung: Von einer Klasse soll nicht mehr als eine Instanz erzeugt werden können. Die Instanziierung soll aber von unterschiedlichen Objekten mehrfach initiiert werden können.

Anwendungsbeispiel: Wir haben eine Klasse, die eine bestimmte (nur einfach vorkommende) Hardware-Ressource repräsentiert (z.B. einen Drucker).

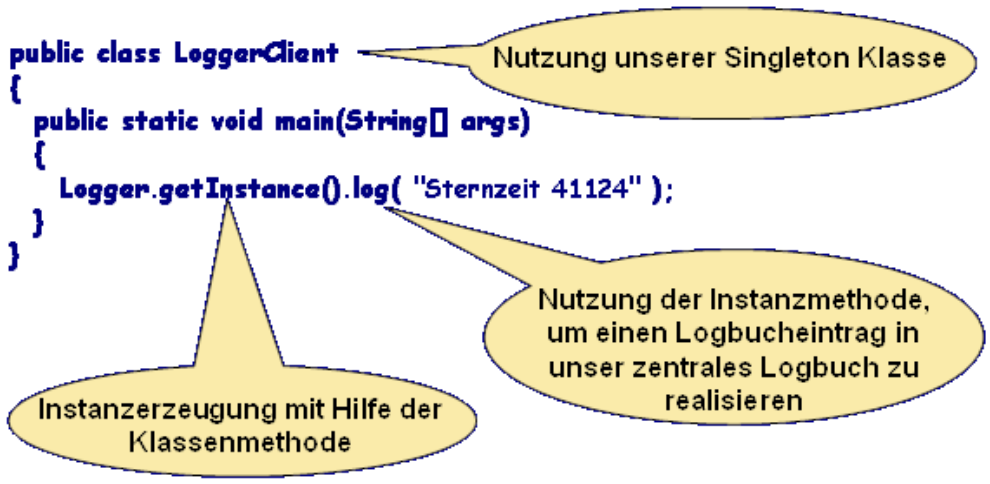
1. Wir deklarieren den Konstruktor als privat, so dass er von außen nicht mehr gerufen werden kann.
2. Wir definieren eine Klassenmethode, mit deren Hilfe eine Instanz erzeugt werden kann.

Beispiel in Java: Wir wollen ein zentrales Logbuch, in dem wir wichtige Ereignisse notieren. Es soll nur ein Logbuch geben, in das alle anderen Objekte eintragen sollen.



Die Klasse Logger ist für das Schreiben eines Logbuchs zuständig und sorgt als Singleton selbst dafür, dass nur eine Instanz existiert.

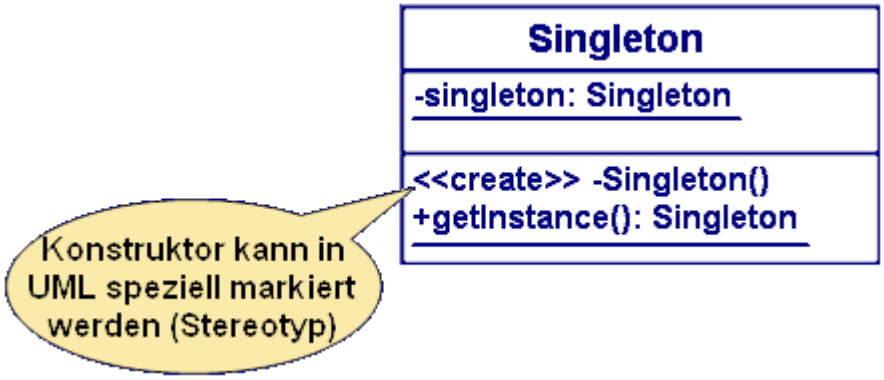
Erzeugung einer Singleton Instanz und Nutzung durch andere Objekte (z.B. LoggerClient):



Prinzip:
Eine Referenz auf das Singleton Objekt wird als Klassenvariable gespeichert.
Fall 1: Es gibt noch keine Instanz
Möchte ein anderes Objekt eine neue Instanz vom Singleton erzeugen, wird diese erzeugt und deren Referenz in die Klassenvariable gelegt.
Fall 2: Es gibt bereits eine Instanz
Möchte ein anderes Objekt eine neue Instanz vom Singleton erzeugen, wird die Referenz in der Klassenvariablen zurückgeliefert statt dass eine weitere Instanz erzeugt wird.

Der Singleton ist damit für die Erzeugung und Verwaltung der einzigen Instanz der Klasse zuständig und bietet einen globalen Zugriff auf diese Instanz über eine Klassenoperation (typischer Name: `getInstance()`).

UML Darstellung des Singleton allgemein:



Ein UML Klassendiagramm zum Beispiel oben würde `Logger` statt `Singleton` als Klassenname enthalten.
Bemerkung: In der UML Darstellung ist der private Konstruktor speziell als Konstruktor gekennzeichnet. Dazu wurde ein sogenannter Stereotyp verwendet (`<<create>>`).

Unser Singleton stellt ein sogenanntes Entwurfsmuster (Design Pattern) dar.
Er dient als bewährte Vorlage (Muster) zur Lösung für ein bestimmtes Problem.
Da Singleton für die (kontrollierte) Erzeugung von Instanzen zuständig ist, wird es in die Kategorie der Erzeugungsmuster (engl. Creational Patterns) eingruppiert.

Bemerkung:

- Das Objektverantwortlichkeitsprinzip kommt hier zum Einsatz: Der Singleton ist selbst dafür verantwortlich, dass nur eine Instanz vorhanden ist. Wir benötigen also keine außenstehende Kontrollinstanz.
- Varianz des Singleton: Multiton (nur eine ganz bestimmte Anzahl von Instanzen einer Klasse ist maximal erlaubt)
- Ähnliche Muster sind z.B. Factory oder Factory Method
- Vorsicht beim Einsatz von Singleton!
  Durch die Klassenvariablen erhalten wir bei starkem Einsatz ein Äquivalent zu globalen Variablen und der Entwurf wird schwieriger zu überblicken. Klasseneigenschaften haben einen schlechten Einfluß bei der Umsetzung des Lokalitätsprinzips.