



Zusammenfassung Skript IOBP01

Grundlagen der objektorientierten Programmierung mit Java (IU Internationale Hochschule)

Einführung in die objektorientierte Programmierung mit Java

1 Einführung in die objektorientierte Systementwicklung

1.1 Objektorientierung als Sichtweise auf komplexe Systeme

Objektorientierung (kurz: OO)-> eine Sichtweise auf komplexe Systeme die als Zusammenspiel von abstrakten oder realen Einheiten (Objekten)

- ➔ **Keine Programmiersprache, Datenbank, etc.**
- ➔ Programmierparadigma und ist die Basis von fast allen Entwicklungsprojekten für betr. Informationssysteme

Übersicht der geschichtl. Entwicklung von Programmierkonzepten:

- **Maschinencode** -> heute keine Verwendung mehr
- **Assemblercode** -> Steuerung für elektrotechnische Geräte (Lüftung, Motor...), reaktive Systeme (Sensorsysteme), hardwarenahe Programmierung
- **Imperative Programmierung** -> kleine Programme zur Lösung einfacher Aufgaben idR in speziellen Programmiersprachen
- **Strukturierte Programmierung** -> einfache Webanwendungen, technische Steuerkomponente
- **Objektorientierte Programmierung** -> große und komplexe industrielle Softwaresysteme
- **Komponentenbasierte Entwicklung** -> Wiederverwendung von bereits programmierten Funktionen, einzelne Komponente sind idR objektorientiert
- **Modellgetriebene Entwicklung** -> Wiederverwendung von einfacher Anpassbarkeit von Querschnittsfunktionen (z.B. Log-In Mechanismen), der Programmcode kann den Konzepten der Objektorientierung entsprechen

1.2 Das Objekt als Grundkonzept der Objektorientierung

Ein Objekt ist ein Bestandteil eines Systems -> ein Objektorientiertes Programm besteht nahezu ausschließlich aus Objekten (Objekte bei Ausführung in Hauptspeicher des Rechners)

Objekte bestehen aus:

- Attributen (speichern Informationen) sind vor direktem Zugriff von anderen Objekten versteckt und geschützt (**Prinzip der Datenkapselung**)
- Methoden (lesen und ändern von Attributen eines Objektes) -> anhand von Methoden können andere Objekte auf die Attribute zugreifen und damit zu arbeiten.
 - ➔ Ein Objekt greift auf ein anderes Objekt zu in dem es eine Methode auf dem Objekt aufruft

1.3 Phasen im objektorientierten Entwicklungsprozess

Ein Softwareentwicklungsprozess (SW-Prozess) hat das Ziel ein lauffähiges System, das die Anforderungen der Auftraggeber in Hinblick auf Funktionalität, Qualität und Entwicklungsaufwand erfüllt.

Im objektorientierten Entwicklungsprozess werden drei Phasen unterschieden:

1. Objektorientierte Analyse (OOA):

- Es wird bestimmt was ein System tun soll mit dem Ziel ein umfangreiches Verständnis der fachlichen Zusammenhänge des Systems zu haben.
->hierzu werden Objekte der realen Welt in einem fachlichen Modell nachgebildet und nur relevante Attribute berücksichtigt
 - Dieses Analysemodell dient der Kommunikation zwischen Entwicklern und Auftraggebern
 - Ausgangspunkt des objektorientierten Designs
2. Objektorientiertes Design (OOD):
- Das Analysemodell wird um technische Informationen erweitert, sodass der Programmierer in der Lage ist mit Hilfe des Designs den Programmcode zu implementieren
 - Es wird festgelegt, welche Arten von Objekten es geben soll und welche Attribute und Methoden sie haben und auf welche Art die Objekte kooperieren.
3. Objektorientierte Programmierung (OOP):
- Hier wird das System Design in funktionierenden Programmcode übersetzt.
 - Analyse und Design sind wichtige Vorarbeiten

1.4 Grundprinzipien der objektorientierten Systementwicklung

Vorteile bei der Objektorientierung:

- Weniger Probleme bei Skalierung
 - Höhere Stabilität
 - einfache Erweiterbarkeit
 - bessere Testbarkeit
 - bessere Wartbarkeit
- ➔ Objektorientierung ist ein Mittel, um die Komplexität von IT-Systemen in den Griff zu bekommen

Beispiel anhand einer Kaffeemaschine dazu auf Seite 17-19

- Jedes Objekt ist für eine bestimmte Funktion zuständig und einzeln ersetzbar
(Kapselung)
- Funktionierendes System „Kaffeemaschine“ durch Kooperation von einzelnen Objekten

2 Einführung in die objektorientierte Modellierung

2.1 Strukturieren von Problemen mit Klassen

- Objekte sind für sich handlungsfähige Einheiten -> werden aus der realen Welt in Objekte der Programmiersprache nachgebildet (nur die wichtigen Attribute werden übernommen)
- Eine Klasse liefert die Struktur, die für die Bildung von Objekten erforderlich ist

- ➔ Alle aus einer Klasse erzeugten Objekte haben die gleiche Struktur (selben Attributen und Methoden) Bsp.: Word-Vorlage = Klasse, daraus erstelltes Dokument = ein Objekt
- ➔ Aus welchen Klassen ein System besteht wird in objektorientiertem Design festgelegt
- Zentrale Aktivität bei der OOA und OOD ist das Identifizieren und Beschreiben von Klassen

2.2 Identifizieren von Klassen

Etablierte Vorgehensweise zur Identifizierung von Klassen:

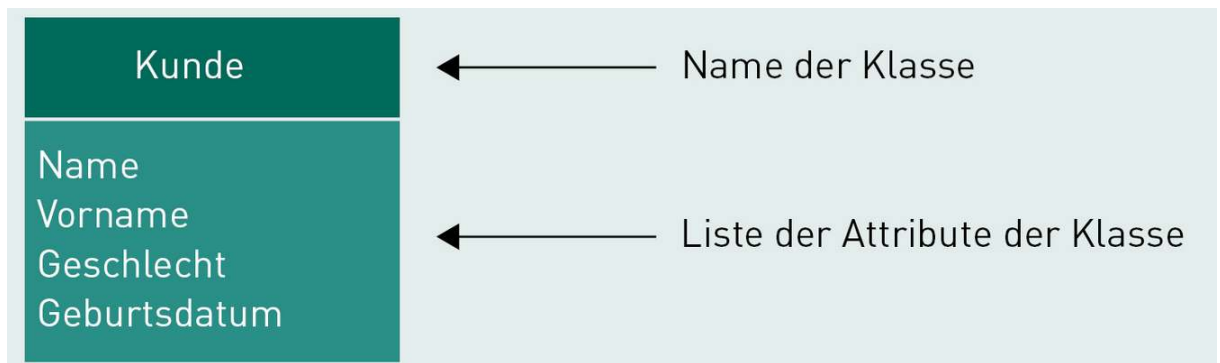
1. Alle Hauptwörter der Aufgabenstellung/Anforderungen werden als Kandidaten für Klassen notiert
2. Es wird geprüft, ob die Hauptwörter durch weitere Hauptwörter beschrieben werden oder ob sie Beziehungen zu anderen Hauptwörtern haben. Trifft eins von beiden zu wird das Hauptwort als Klasse modelliert
3. Wenn ein Hauptwort verwendet wird, um ein anderes Hauptwort zu detaillieren, wird es als Attribut modelliert
4. Alle anderen Wörter, die keine Klasse und Attribut sind, werden nochmal geprüft oder ggf. gestrichen

➔ Klassen werden immer in Form eines Rechtecks dargestellt



2.3 Attribute als Eigenschaften von Klassen

- Attribute sind statische Elemente von Klassen -> es können konkrete Werte zu dem Objekt gespeichert werden
- Die Werte aller Attribute eines Objektes beschreiben den Zustand
- Attribute werden als Rechteck unter dem Namen der Klasse dargestellt



- Bei der Modellierung der Attribute können folgende Eigenschaften festgelegt werden (im Analysemodell ist der Name Pflicht, die fehlenden Eigenschaften werden im Laufe des SW-Prozesses festgelegt):

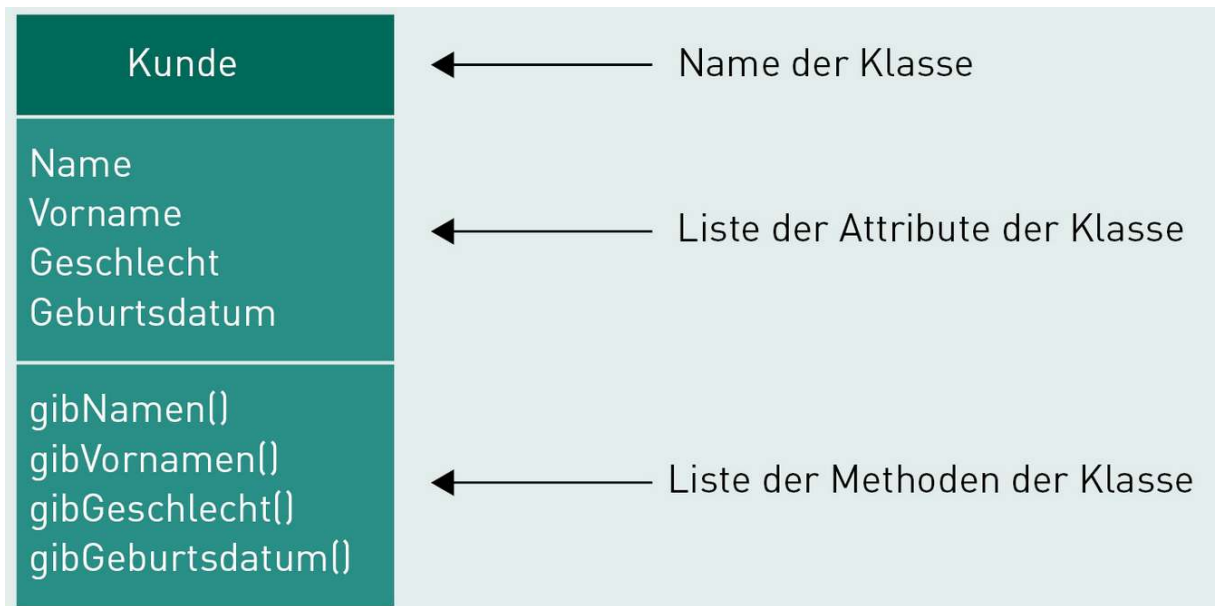
Name	Name des Attributs	Vorname
Datentyp	Legt fest, wie die Werte zu dem Attribut aussehen, also ob es z. B. eine Zahl, eine Zeichenkette oder ein Datum ist	Date (Datum) String (Zeichenkette) Integer (Ganze Zahl)
Konstante (ja/nein)	Legt fest, ob sich der Wert des Attributs ändern darf oder nicht	Gerundete Kreiszahl Pi: 3,1415
Defaultwert	Legt den voreingestellten Wert des Attributes fest	2010-01-01

2.4 Methoden als Funktionen von Klassen

- Sind dynamische Elemente von Klassen -> enthalten Algorithmen, Anweisungen etc. Mit denen Werte erstellt, berechnet verändert oder gelöscht werden können. Ergebnisse von Methoden können in Attributen der Klasse oder in neuen Objekten gespeichert werden
- Mit Methoden wird das Verhalten von Objekten beschrieben
- Bei Modellierung von Methoden können folgende Eigenschaften festgelegt werden (im Analysemodell ist der Name Pflicht, die fehlenden Eigenschaften werden im Laufe des SW-Prozesses festgelegt):

Name	Name der Methode	getName()
Parameter	Benötigte Objekte und Werte, die zur Abarbeitung der Methode erforderlich sind	(Date geburtsdatum) (String name, String vorname) (Integer zahl1, Integer zahl2)
Rückgabewert	Angabe des Datentyps des Objektes, in dem das Ergebnis der Methode gespeichert wird	Date (Datum) String (Zeichenkette) Integer (Ganze Zahl)

- Methoden werden auch in einem Rechteck unterhalb der Attribute dargestellt



2.5 Beziehungen zwischen Klassen







- Alle Elemente in einem objektorientierten System werden durch Klassen programmiert -> ein Zusammenspiel zwischen Klassen ist nur möglich, wenn Beziehungen zwischen Klassen definiert werden

„hat/kennt“	Dieser Beziehungstyp drückt aus, dass eine Klasse eine andere Klasse „hat“ oder „kennt“.	Ein Versicherungsnehmer hat Kinder. Ein Vertrag hat Versicherungsbedingungen. Ein Kalender hat Monate. Ein Verkäufer kennt seine Kunden.
„besteht aus“	Dieser Beziehungstyp drückt aus, dass eine Klasse ein Bestandteil einer anderen Klasse ist. Er wird genutzt, wenn eine Klasse ein größeres Konstrukt ist, dessen Elemente nicht durch einfache Attribute beschrieben werden können.	Ein Auto besteht aus einem Motor, 4 Rädern, 3 Türen, 1 Getriebe und 2 Sitzen. Ein Haus besteht aus einem Dach, 23 Fenstern, 2 Türen, 6 Räumen und 1 Treppenhaus.
„ist ein“	Dieser Beziehungstyp drückt aus, dass eine Klasse A von der Art her eine Klasse B ist, aber eine spezifischere Bedeutung hat und sich ggf. um bestimmte Attribute und Methoden von Klasse B unterscheidet.	Ein Pkw ist ein Auto. Ein Lkw ist ein Auto. Ein Kunde ist eine Person. Ein Buch ist ein Artikel. Ein Igel ist ein Säugetier.

- Beziehungen zwischen Klassen werden wie folgt dargestellt:

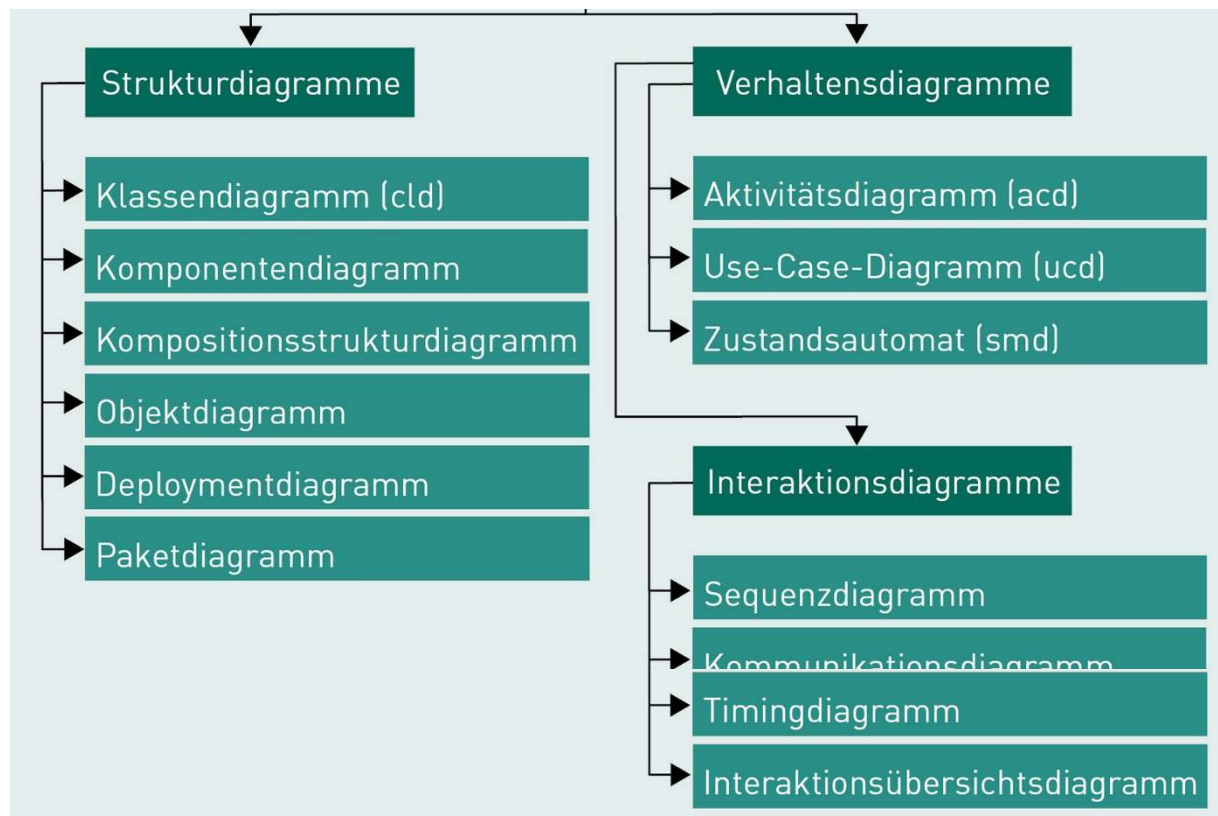
<p>Durchgezogene Linie, ohne Pfeilspitze und ohne Beschriftung</p>	Vertrag und Adresse stehen in einer nicht näher beschriebenen Beziehung zueinander.
<p>Durchgezogene Linie, Beschriftung der Linie mit einem Namen für die Beziehung</p>	Vertrag und Adresse sind über eine benannte Assoziation verbunden; die beide Klassen sind verbunden durch die Beziehung „Rechnungsanschrift“.
<p>Durchgezogene Linie mit einer Pfeilspitze, ggf. Benennung der Beziehung</p>	Durch die Pfeilspitze wird eine Navigationsrichtung vorgeben: vom Vertrag zur Adresse. Das bedeutet dass der Vertrag eine Adresse kennt und man sich vom Vertrag zur Adresse durchhangeln kann. Jedo nicht von der Adresse zum Vertrag: Ein Vertrag kennt seine Adresse, aber die Adresse weiß nichts vor und über die Existenz des Vertrags.
<p>An den Enden der Beziehung werden Multiplizitäten modelliert und damit Aussagen über die Anzahl der assoziierten Objekte gemacht.</p>	Ein Vertrag hat genau eine Rechnungsanschrift, eine Adresse kann jedoch die Rechnungsanschrift zu mindestens 1 aber maximal beliebig vielen Verträgen sein. (Detaillierte Erklärung dazu siehe unten.)

- Beziehungen können auch mit Multiplizitäten näher erläutert werden. (links = Untergrenze; rechts= Obergrenze):

0..1	Optionale Assoziation	 <p>Zu einem Auto gehören 0..1 Anhänger. Zu einem Anhänger gehören 0..1 Autos.</p>
1	Obligatorische Assoziation	 <p>Zu einem Auto gehört genau 1 Fahrer. Ein Fahrer gehört zu genau 1 Auto.</p>
0..*	Optional beliebig	 <p>Ein Student kann 0..* Kurse belegen. Ein Kurs kann von 1..* Studenten belegt sein.</p>
1..*	Beliebig, aber mindestens 1	 <p>Ein Tutor kann 1..* Kurse machen. Ein Kurs wird von genau 1 Tutor durchgeführt.</p>
n..m	Mindestens n und maximal m	 <p>Ein Auto hat mindestens 3 und maximal 4 Türen. Eine Tür gehört zu genau 1 Auto.</p>
	Keine Angabe entspricht 1 (sollte aber vermieden werden, um Missverständnisse zu vermeiden)	 <p>Ein Auto hat genau 1 Motor. Ein Motor gehört zu genau 1 Auto.</p>

2.6 Unified Modeling Language (UML)

- Um die Ergebnisse der Phase OOA und OOD zu dokumentieren werden die identifizierten Klassen und die Beziehungen in einer Modellierungssprache dokumentiert
- Es lassen sich Diagramme zur Beschreibung von der Struktur und Verhalten unterscheiden
 - Strukturdiagramme:** stellen dar, woraus ein System besteht (Aufbau, Elemente, Zusammensetzung und Schnittstellen)
 - Verhaltensdiagramme:** stellen dar, was in einem System abläuft
- Es gibt ca. 13 verschiedene Diagrammtypen siehe folgend:



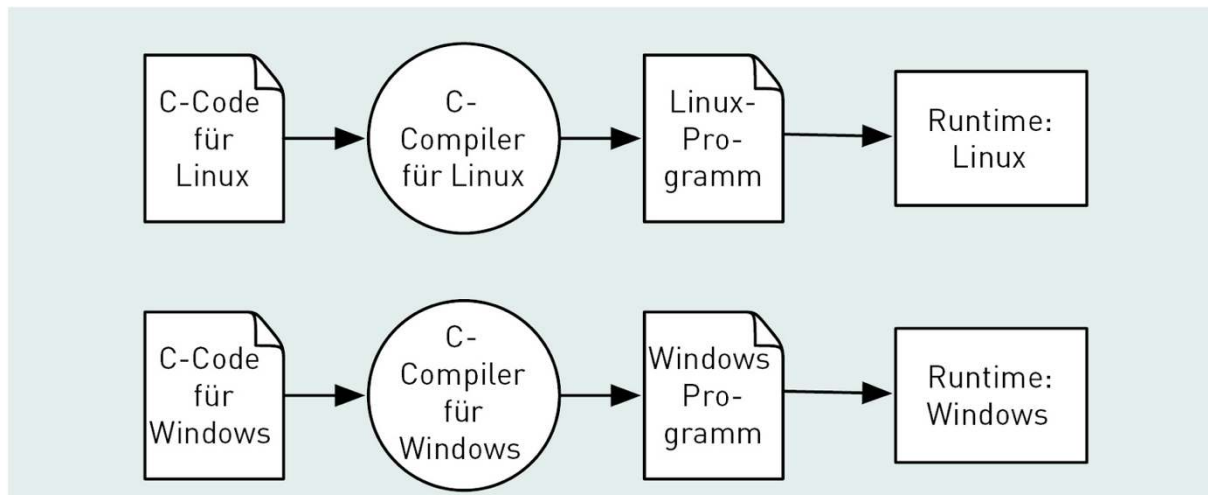
- Klassen mit Methoden und Attributen werden mit dem UML Klassendiagramm modelliert
- Objektdiagramme sind eine Spezialform der Klassendiagramme -> es werden konkrete Ausprägungen von Klassen dargestellt (mit Werten)
- Unterschiede von Objekten zu Klassen:
 1. Jedes Objekt ist eindeutig identifizierbar mit einer ID (notiert links von Klassennamen gefolgt vom Doppelpunkt)
 2. Zu jedem Attribut eines Objektes wird ein konkreter Wert dargestellt.

3 Programmieren von Klassen in Java

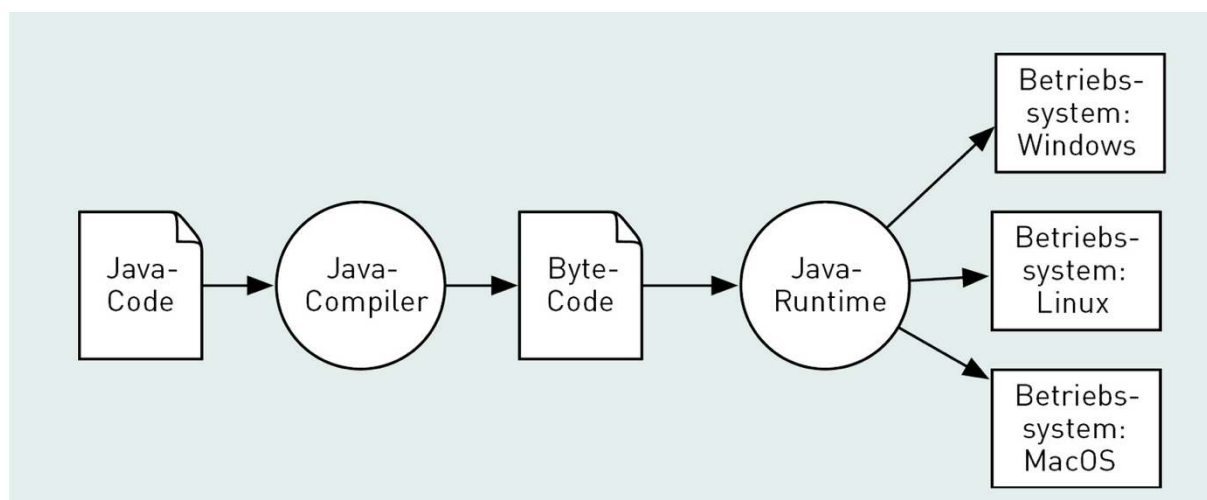
3.1 Einführung in die Programmiersprache Java

Grundsätzlich gibt es drei verschiedene Aktivitäten beim Programmieren:

1. Produzieren von Programmcode -> kann mit Texteditor erstellt werden
 2. Kompilieren des Programmcodes -> Software benötigt zum übersetzten in Maschinencode (Compiler gibt es für jede Programmiersprache)
 3. Ausführen des Programms -> danach kann das Programm in einer Runtime ausgeführt werden
- Da sich Funktionen und Eigenschaften bei Betriebssystemen unterscheiden müssen Programme für jedes Betriebssystem neu programmiert bzw. kompiliert werden.



- Java ist eine plattform-unabhängige Programmiersprache -> muss nicht für jedes Betriebssystem angepasst werden.
- Java unterscheidet sich zu anderen Programmiersprachen, da bevor ein Programm ausgeführt werden kann eine Java Runtime Environment(JRE) installiert werden muss
 - ➔ Der Java Compiler übersetzt den Programmcode in einen Bytecode (eine Form eines Java Programms nach dem kompilieren)
 - ➔ JRE ist für alle Betriebssysteme verfügbar

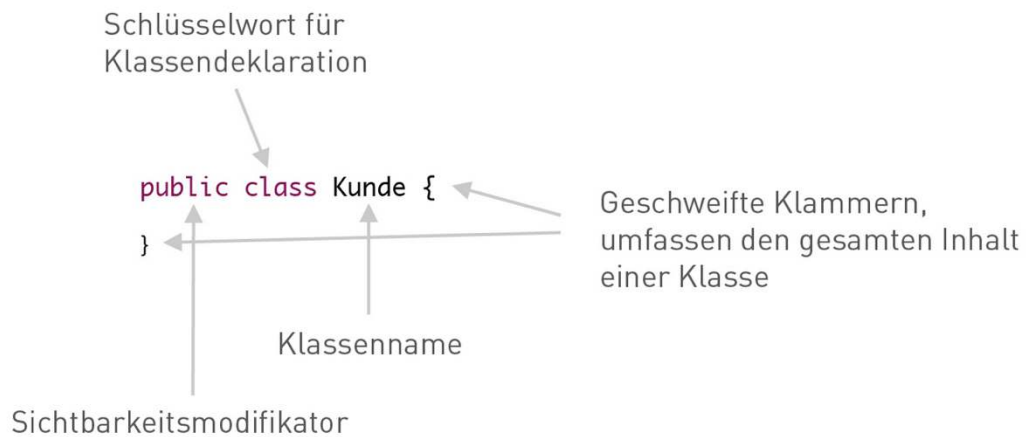


- Die JRE besteht aus:
 - ➔ Java Virtual Machine (JVM) -> Software die den Bytecode interpretiert und im Betriebssystem startet und ausführt
 - ➔ Java Klassenbibliothek -> stellt schon vorhandene Funktionen zur Verfügung
- Zur Entwicklung werden Java Software Development Kits (Java SDK) benötigt bestehend aus: Compiler, Java Virtual Machine, der Klassenbibliothek und Hilfsprogrammen

3.2 Grundelemente einer Klasse in Java

- Eine Klasse hat einen eindeutigen Namen und besteht aus Attributen und Methoden

- ➔ Der Programmcode wird in einer Datei gespeichert mit der Endung .java und der Bytecode wird vom Compiler in einer Datei mit der Endung .class gespeichert
- Mindestanforderungen für eine Klasse sind:
 - ➔ Sichtbarkeitsmodifikator, Schlüsselwort „class“, der Name, ein paar {}



Grundelemente einer Klasse näher erklärt:

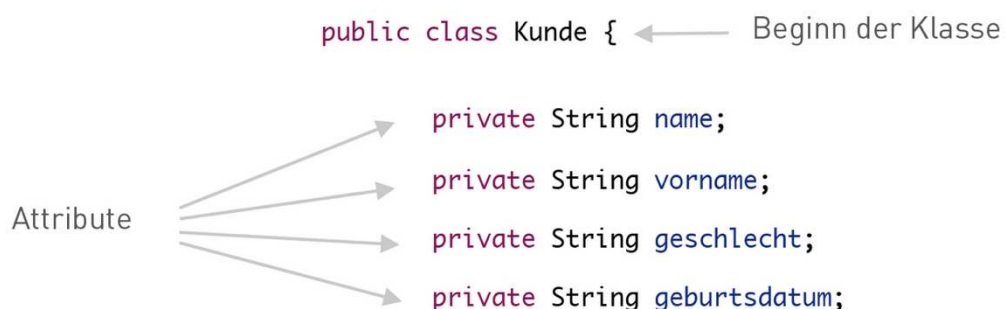
Sichtbarkeitsmodifikator	Legt die Sichtbarkeit der Klasse für andere Klassen fest (Details siehe Lernzyklus 4.5)	public
Schlüsselwort für die Klassendeklaration	Zeigt dem Java-Compiler an, dass im Folgenden eine Java-Klasse programmiert ist	class
Klassenname	Legt den Namen für die Klasse in Java fest und wird als Dateiname verwendet. Die Klasse „Kunde“ wird in einer Textdatei „Kunde.java“ programmiert	Kunde
Geschweifte Klammern	Markieren den Inhalt einer Klasse (Attribute und Methoden); alles was innerhalb der Klammern steht, gehört zu einer Klasse	{ ... }

Für den Namen einer Klasse gelten folgende Vorgaben:

- Beginnend mit Großbuchstaben
- Nur Unicodezeichen
- Theoretisch beliebig lang (begrenzt durch max. Länge des Dateinamens)
- Darf kein Schlüsselwort sein
- Wenn mehrere Wörter werden diese ohne Trennzeichen zusammengeschrieben

3.3 Attribute in Java

- Um Attribute einer Klasse zu erstellen sind mindestens folgende Elemente nötig:
 - ➔ Sichtbarkeitsmodifikator, der Datentyp und der Attributname



- Den Attributen können direkt bei der Erstellung Werte zugewiesen werden (=Defaultwert)

Grundelemente von Attributen näher erklärt:

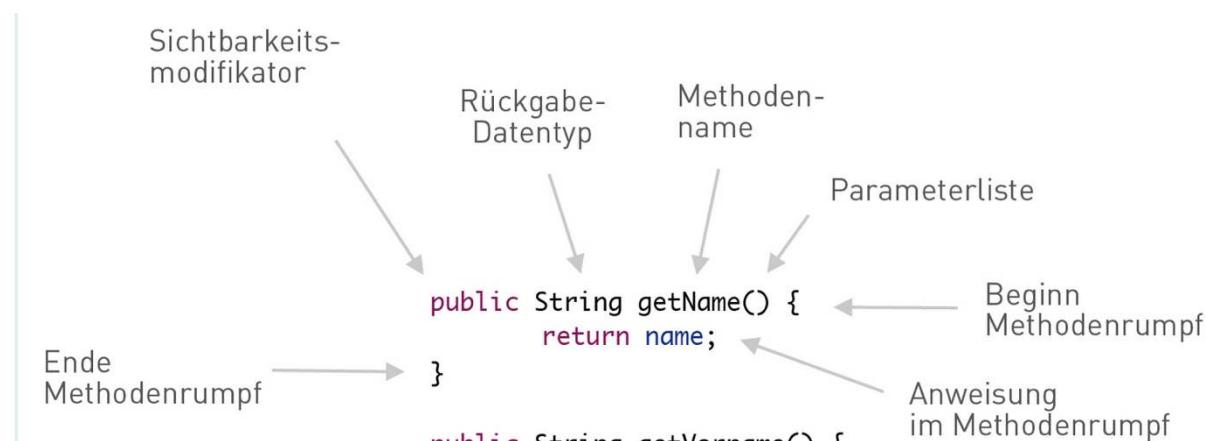
Sichtbarkeitsmodifikator	Legt die Sichtbarkeit des Attributs für andere Klassen fest (Details siehe Lernzyklus 4.5)	private
Datentyp des Attributs	Legt den Datentyp des Attributs fest und bestimmt damit Anzahl und Art der Werte, die in dem Attribut gespeichert werden können. Ein Datentyp ist entweder ein primitiver Datentyp (Details siehe Lernzyklus 4.1) oder eine Klasse, ein sogenannter Referenzdatentyp (z. B. String oder Kunde).	String Kunde
Attributname	Legt den Namen für das Attribut der Klasse fest; jeder Name darf innerhalb einer Klasse nur 1x vergeben werden.	name vorname geschlecht
Defaultwert	Legt den initialen Wert des Attributs fest; dieser Wert wird dem Attribut bei der Erzeugung eines Objektes der Klasse zugewiesen.	= 0 = 1 = false
Semikolon	Markiert das Ende der Attributdeklaration.	;

Für den Namen eines Attributs gelten folgende Vorgaben:

- Beginnt mit Kleinbuchstaben
- Nur Unicode Zeichen
- Theoretisch beliebig lang
- Darf keine Schlüsselworte enthalten
- Groß- und Kleinschreibung wird beachtet
- Bei mehreren Wörtern werden diese ohne Trennzeichen zusammengeschrieben

3.4 Methoden in Java

- Sind dynamische Elemente von Klassen, die es erlauben die Werte von Attributen zu erstellen, verändern und zu löschen
- Andere Objekte können nur über Methoden auf die Attribute eines Objektes zugreifen (**Prinzip der Datenkapselung**)



Folgende Abbildung erklärt die Grundelemente einer Methode näher:

Sichtbarkeitsmodifikator	Legt die Sichtbarkeit der Methode für andere Klassen fest (Details siehe Lernzyklus 4.5).	public
Rückgabe-Datentyp der Methode	Angabe des Datentyps des Objektes, in dem das Ergebnis der Methode nach Abarbeitung des Methodenrumpfes ausgegeben wird. Dabei werden ein primitiver Datentyp (Details siehe Lernzyklus 4.1) oder eine Klasse (z. B. String oder Kunde) angegeben. Für den Fall, dass die Methode kein Ergebnis ausgibt, wird void als Rückgabe-Datentyp festgelegt.	String Kunde void
Methodenname	Legt den Namen für die Methode fest; ein Methodenname darf innerhalb einer Klasse nur dann mehrfach vergeben werden, wenn sich Anzahl bzw. Datentyp der Parameter unterscheiden.	getName() getVorname()
Parameterliste	Liste benötigter Objekte und deren Datentypen, die zur Abarbeitung der Methode erforderlich sind; werden keine Parameter angegeben bleibt die Liste leer.	(Date geburtsdatum) (String name,String vorname) (Integer zahl1,Integer zahl2) ()
Methodenrumpf	Enthält die konkreten Anweisungen, was beim Aufruf der Methode in welcher Reihenfolge getan wird. Jede Anweisung wird mit einem Semikolon ";" beendet, daher ist in Methodenrumpfen das ";" oft das letzte Zeichen einer Programmzeile. Die Anweisungen innerhalb eines Methodenrumpfes werden der Reihe nach von oben nach unten abgearbeitet. Wird für die Methode ein Rückgabe-Datentyp angegeben, beginnt die letzte Anweisung des Methodenrumpfes mit dem Schlüsselwort return.	ergebnis = zahl1 + zahl2; return name;

Für den Namen einer Methode gelten folgende Vorgaben:

- Beginnt mit Kleinbuchstaben
- Nur Unicode Zeichen
- Theoretisch beliebig lang
- Keine Schlüsselwörter
- Groß- und Kleinschreibung wird beachtet
- Bei mehreren Wörtern werden diese ohne Trennzeichen zusammengeschrieben
- Ein weiteres Konzept bei Methoden ist die **Signatur**. Sie besteht aus dem Namen und der Parameterliste und darf in einer Klasse nicht doppelt vorkommen. Methoden können gleich heißen wenn die Parameterliste sich unterscheidet!
- Namen sollten beschreiben, was die Methode tut.
- Zum Auslesen und schreiben von Attributen werden **Getter und Setter-Methoden** verwendet (wenn Attribute private sind, müssen Getter und Setter Methoden implementiert werden -> Getter für boolean -> isPremiumKunde und nicht getPremiumKunde)

```

public class Kunde {
    ...
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    ...
}

```

Attribut →

Getter-Methode

Ausgeben des aktuellen Attributwerts

Setter-Methode

Ändern des aktuellen Attributwerts auf den Wert des Parameters

- Um eine Methode auszuführen muss sie aufgerufen werden -> kann im Methodenrumpf einer anderen Methode derselben Klasse oder aus Methodenrumpfen aus anderen Klassen aufgerufen werden. -> Aufruf erfolgt über den Namen der Methode:

Codebeispiel 1: Aufruf der Methode `getName()` innerhalb der Klasse „Kunde“

▼ Code

```
String id1= getName();
```

Hier wird die Getter-Methode des Attributs `name` aufgerufen, um den Wert von `name` in einer lokalen Variable `id1` (Details zu Variablen siehe Lernzyklus 4.2) zu speichern.

Codebeispiel 2: Aufruf der Methode `setName()` innerhalb der Klasse „Kunde“

▼ Code

```
setName("Lange");
```

Hier wird die Setter-Methode des Attributs `name` aufgerufen, um den aktuellen Wert von `name` auf den Wert „Lange“ zu ändern.

Codebeispiel 3: Aufruf der Methode `getName()` der Klasse „Kunde“ aus einer anderen Klasse heraus

▼ Code

```
String id2= kunde1.getName();
```

Das Objekt, das die Methode `getName()` in Kunde aufrufen will, hat ein Objekt der Klasse „Kunde“ unter dem Variablennamen `kunde1` gespeichert. Mit dem Aufruf `kunde1.getName()` kann gezielt die Methode `getName()` in dem Objekt `kunde1` aufgerufen werden. Der Wert von `kunde1.getName()` wird in der aufrufenden Klasse in der Variable (Details zu Variablen siehe Lernzyklus 4.2) `id2` gespeichert.

Codebeispiel 4: Aufruf der Methode `setName()` aus einer anderen Klasse heraus

▼ Code

```
kunde1.setName("Lange");
```

Das Objekt, das die Methode `setName()` in Kunde aufrufen will, hat ein Objekt der Klasse „Kunde“ unter dem Variablennamen `kunde1` gespeichert. Mit dem Aufruf `kunde1.setName("Lange")` kann gezielt die Methode `setName(String name)` in dem Objekt `kunde1` aufgerufen werden. Über diesen Methodenaufruf kann aus einem anderen Objekt heraus der Wert „Lange“ als Attribut in das Objekt `kunde1` gespeichert werden.

- Überladen von Methoden = wenn mehrere Methoden mit demselben Namen aber unterschiedlichen Parameterlisten in einer Klasse implementiert sind

Beispiel Überladen:

Beispiel

Die folgende Methode soll verwendet werden, um im Online-Shop von Frau Lange einen Artikel in den Warenkorb zu legen. Dafür gibt es eine Methode `zumWarenkorbHinzufuegen`, die als Parameter den entsprechenden Artikel enthält. Der Artikel ist ein Objekt der Klasse „Artikel“:

▼ Code

```
public void zumWarenkorbHinzufuegen (Artikel artikel) {...}
```

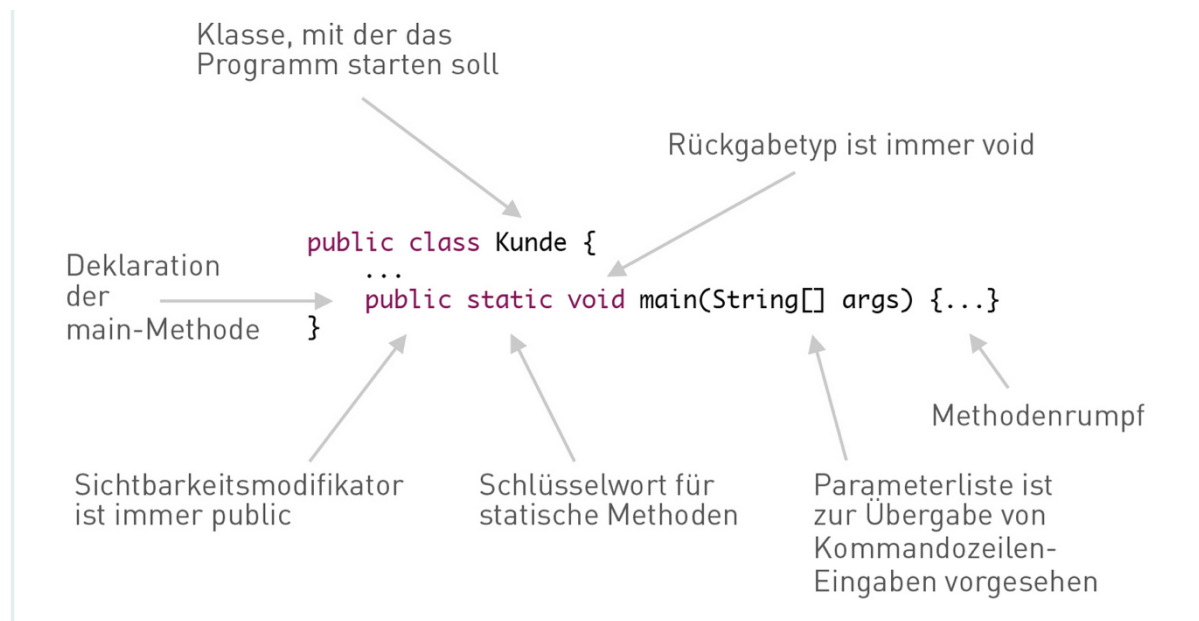
Zusätzlich zu dieser Methode soll es aber auch die Möglichkeit geben, mehrere gleiche Artikel in den Warenkorb zu legen. Dazu wird die Methode `zumWarenkorbHinzufuegen` überladen, indem eine zweite Methode `zumWarenkorbHinzufuegen` implementiert wird, die neben dem Artikel auch die Anzahl der hinzuzufügenden Artikel als Parameter erwartet:

▼ Code

```
public void zumWarenkorbHinzufuegen (Artikel artikel, int anzahl) {...}
```

3.5 main-Methode: Startpunkt eines Java-Programms

- eine Java-Programm ist ein Netz von miteinander kooperierenden Objekten
- für jedes Java-Programm gibt es einen Startpunkt. -> main-Methode
 - ➔ das Erzeugen aller von Programm benötigten Objekte startet in der Methode



Folgende Elemente der main-Methode sollten ohne Änderung übernommen werden:

- Sichtbarkeitsmodifikator -> **public**
- Deklaration als statische Methode -> **static**
- Festlegung, dass es keine Rückgabewert gibt -> **void**
- Name der Methode -> **main**
- Und die Parameterliste -> **String args[]**

➔ Nur der Methodenrumpf wird angepasst

4 Java Sprachkonstrukte

4.1 Primitive Datentypen

- Sind einfache Datentypen die angeben, welche Art von Information in einem Attribut gespeichert werden können:

Wahrheitswerte	boolean	Kann entweder true oder false sein. Weitere Werte sind nicht erlaubt.	true false
Ganze Zahl	byte	8Bit-Wertebereich von -128 bis 127	123, 0, 23, 120
	short	16Bit-Wertebereich von -32.768 bis 32.767	-23000, 0, 13231
	int	32Bit-Wertebereich von -2.147.483.648 bis 2.147.483.647	-12332123, 234, 1102379239
	long	64Bit-Wertebereich von -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.808	8347829790645, 13, 0, 34879, 789234789274
Fließkommazahl	float	32Bit-Wertebereich von $1,40239846 \cdot 10^{-45}$ bis $3,40282347 \cdot 10^{38}$	1.87236f (Einfache Zahl mit Komma und angehängtem „f“) -3.938e12f (-3.938*10 ¹² mit angehängtem „f“)
	double	64Bit-Wertebereich von $4,94065645841246544 \cdot 10^{-324}$ bis $1,79769131486231570 \cdot 10^{308}$	1.87236d (Einfache Zahl mit Komma und angehängtem „d“) -3.938e120d (-3.938*10 ¹²⁰ mit angehängtem „d“)
Schriftzeichen (Character)	char	Ein einzelnes Unicode-Zeichen. Umfasst neben Ziffern, Buchstaben und Symbolen auch Steuerzeichen wie Leerzeichen, Tabulator oder Zeilenumbruch.	'A' (= Buchstabe A), '2' (= Ziffer 2), '\n' (= Steuerzeichen Zeilenumbruch)
Zeichenketten	String	Wird zum Speichern von beliebig langen Zeichenketten verwendet. Ist kein primitiver Datentyp. Die Klasse String bietet viele bereits vorhandene Methoden für die Bearbeitung von Zeichenketten.	"online", "Herr Koch", "Was nicht passt wird passend gemacht", ""

4.2 Variablen

- Variablen bieten die Möglichkeit konkrete Werte im Speicher der Anwendung abzulegen
- Zur Deklaration einer Variablen muss der Name und der Datentyp angegeben werden
-> abgeschlossen mit ;
 - ➔ Benötigen keinen Sichtbarkeitsmodifikator, da sie nur im Methodenrumpf verwendet werden
- Deklaration und Zuweisung eines Wertes kann direkt oder in zwei Schritten geschehen. Hier direkt:

```
public void rechnerei() {  
    int zahl1 = 234;  
    byte zahl2 = 123;  
    float zahl3 = 123.4f;  
    char einZeichen = 'E';  
    String zeichenkette = "Das ist eine Zeichenkette";  
}
```

4.3 Operatoren und Ausdrücke

- Operatoren ermöglichen das Rechnen und Verändern von in Variablen gespeicherten Werten

Die wichtigsten Arithmetischen Operatoren sind:

Arithmetische Addition; der Ergebnistyp der Berechnung entspricht dem des Operanden mit dem größten Wertebereich (z. B. int+int=int, int+long=long;)	+	Ganze Zahlen, Fließkommazahlen	int c,d,e; c = 3; d = 5; e = c + d; int x; long y,z; x = 3; y = 4; z = x + y;
Arithmetische Subtraktion	-	Ganze Zahlen, Fließkommazahlen	int e = 3; float f = 4; float g; g = e - f;
Arithmetische Multiplikation	*	Ganze Zahlen, Fließkommazahlen	int c,d; c = 3; d = c * 4;
Arithmetische Division; berechnet Quotienten aus Dividend und Divisor	/	Ganze Zahlen: Sind beide Operanden ganze Zahlen, wird vom Ergebnis der Teil nach dem Komma abgeschnitten. Fließkommazahlen: Ist mindestens ein Operand eine Fließkommazahl, wird das Ergebnis auch eine Fließkommazahl und nicht gerundet.	int h = 4; int i = 3; int j; j = h/i; int o = 4; double p = 3; double q; q = o/p;
Rest (auch: Restwert-Operator); berechnet den Rest der arithmetischen Division	%	Ganze Zahlen, Fließkommazahlen	int k = 11; int l = 5; int m; m = k % l;

- Logische Operatoren werten Ausdrücke zu true oder false aus-> häufig eingesetzt zur Steuerung von Kontrollstrukturen:

Logisches Komplement (Negation); ändert den Wahrheitswert des Operanden	!	boolean	boolean b1; boolean b2 = false; b1 = !b2;
Logisches UND; liefert true, wenn beide Operanden true sind	&&	boolean	boolean b3; boolean b4 = true; boolean b5 = true; b3 = b4 && b5;
Logisches ODER; liefert true, wenn einer der beiden Operanden true ist		boolean	boolean b6; boolean b7 = false; boolean b8 = true; b6 = b7 b8;
Exklusiv-ODER; liefert true, wenn ein Operand true und ein Operand false ist	^	boolean	boolean b9; boolean b10 = false; boolean b11 = true; b9 = b10 ^ b11;

- Vergleichsoperatoren vergleichen Ausdrücke miteinander und liefern als Ergebnistyp boolean-> häufig zur Steuerung von Kontrollstrukturen verwendet:

Gleichheit	Primitive Datentypen: liefert true, wenn die Werte der Operanden gleich sind	==	Primitive Datentypen	int z1, z2; boolean e1; z1 = 3; z2 = 3; e1 = z1 == z2;
	Referenzdatentypen: liefert true, wenn in beiden Operanden die Referenz auf dasselbe Objekt enthalten ist		Referenzdatentypen	Kunde kunde1, kunde2; boolean e2; kunde1 = new Kunde(); kunde2 = kunde1; e2 = kunde1 == kunde2;
Ungleichheit	Primitive Datentypen: liefert true, wenn die Werte der Operanden nicht gleich sind	!=	Primitive Datentypen	int z3, z4; boolean e3; z3 = 4; z4 = 3; e3 = z3 != z4;
	Referenzdatentypen: liefert true, wenn in beiden Operanden die unterschiedlichen Referenzen enthalten sind		Referenzdatentypen	Kunde kunde3, kunde4; boolean e4; kunde3 = new Kunde(); kunde4 = new Kunde(); e4 = kunde3 != kunde4;
Kleiner als liefert true, wenn der Wert des linken Operanden kleiner ist verglichen mit dem Wert des rechten Operanden		<	Ganze Zahlen, Fließkommazahlen	int z5, z6; boolean e5; z5 = 4; z6 = 5; e5 = z5 < z6;
Kleiner gleich liefert true, wenn der Wert des linken Operanden kleiner oder gleich ist verglichen mit dem Wert des rechten Operanden		<=	Ganze Zahlen, Fließkommazahlen	int z5, z6; boolean e5; z5 = 4; z6 = 5; e5 = z5 <= z6;
Größer als liefert true, wenn der Wert des linken Operanden größer ist verglichen mit dem Wert des rechten Operanden		>	Ganze Zahlen, Fließkommazahlen	int z7, z8; boolean e6; z7 = 6; z8 = 5; e6 = z7 > z8;
Größer gleich liefert true, wenn der Wert des linken Operanden größer oder gleich ist verglichen mit dem Wert des rechten Operanden		>=	Ganze Zahlen, Fließkommazahlen	int z7, z8; boolean e6; z7 = 6; z8 = 5; e6 = z7 >= z8;
Typvergleich liefert true, wenn der Datentyp des linken Operanden gleich dem gegebenen Datentypen im rechten Operanden ist		instanceof	Referenzdatentypen	Kunde k1; boolean e7; k1 = new Kunde(); e7 = k1 instanceof Kunde;

- ==(Gleichheit) und !=(Ungleichheit) sind besondere Prüfoperatoren, da hier die Operanden festlegen, ob Werte oder Referenzen verglichen werden.
- Wenn Referenzen verglichen werden, prüft der Operator nicht die Werte sondern, ob auf das gleiche Objekt verwiesen wird.
- Bei Verkettung von Strings wird der Operator + verwendet.

4.4 Kontrollstrukturen

- Kontrollstrukturen sind Elemente einer Programmiersprache zur bedingten oder mehrfachen Ausführung von Anweisungen.

➔ Die wichtigsten sind: bedingte Verzweigungen (if-else) und Schleifen (for, while, do-while)

Die bedingte Verzweigung:

- Wird verwendet um die konkrete Stelle mit der das Programm fortgesetzt wird anhand einer Bedingung prüft:

▼ Code

```
if (Bedingung) {  
    Anweisung1;  
}  
  
else {  
    Anweisung2;  
}
```

-> die Bedingung muss ein Ausdruck sein, der zu true oder false ausgewertet werden kann. -> else - Teil ist optional.

Die erweiterte if-else-Verzweigung:

- Hier wird nicht nur eine Bedingung geprüft, bevor der else Block abgearbeitet wird, sondern mehrere sich gegenseitig ausschließende Bedingungen.

```
if (Bedingung1) {  
    Anweisung1;  
}  
  
else if (Bedingung2) {  
    Anweisung2;  
}  
  
else {  
    Anweisung3;  
}
```

Schleifen:

- Ermöglichen die mehrfache Ausführung von gleichen Anweisungen hintereinander
Drei verschiedene Schleifenarten: While-Schleife, do-while Schleife, for- Schleife

While Schleife:

- Prüft zuerst die Schleifenbedingung, wenn die Bedingung erfüllt ist werden die Anweisungen der Schleife ausgeführt. Andernfalls werden die Anweisungen übersprungen. Nach der Ausführung wird die Bedingung erneut geprüft, wenn false
➔ Ist die Schleife beendet

▼ Code

```
while (Bedingung) {  
    Anweisungen;  
}
```

- Wird auch kopfgesteuerte Schleife genannt, denn vor dem ersten Durchlauf wird die Bedingung geprüft.

Do-while-Schleife:

- Ist eine Fußgesteuerte while-Schleife, die Anweisungen werden in jedem Fall einmal durchlaufen und dann wird die Schleifenbedingung geprüft-> wenn true wird die Anweisung wiederholt

```
do {  
    Anweisungen;  
} while (Bedingung)
```

For-Schleife:

- Elemente des Schleifenkopfes sind die Initialisierung, Bedingungsprüfung und das Ändern der Zählvariablen
- Kopfgesteuert

```
for (Initialisierung; Bedingung; Schleifenfortschaltung) {  
    Anweisungen;  
}
```

Das Ablaufschema ist wie folgt:

1. Ausführen der Anweisung der initialisierung
 2. Prüfung der Bedingung
 3. Auswertung der Bedingung
 - ➔ Wenn true dann Ausführung aller Anweisungen der For-Schleife
 - ➔ Wenn false dann Abbruch und keine Ausführung
 4. Ausführung der Anweisung der Schleifenfortschaltung
 5. Weiter mit Punkt 2
- Es gibt auch verschachtelte For-Schleifen, dabei kann können die die Variablen, die in der äußeren For-Schleife deklariert wurden auch in der inneren Schleife genutzt werden. Andersherum nicht

4.5 Pakete und Sichtbarkeitsmodifikatoren

- Zur logischen Strukturierung werden die Java-Klassen in Paketen strukturiert -> Klassen mit Abhängigkeiten und mit ähnlichen Funktionen werden in gleichen Paketen zugeordnet.
 - ➔ Hat Auswirkungen auf den Speicherort der Klasse und die Zugriffsberechtigung anderer Klassen
- Pakete können auch selbst Pakete enthalten
- Klassennamen müssen innerhalb eines Paketes eindeutig sein
 - ➔ Der qualifizierte Name ergibt sich aus dem Paket der Klasse und dem Klassennamen
- Mit Sichtbarkeitsmodifikatoren wird die Zugriffsmöglichkeit von Klassen, Attributen und Methoden von anderen Klassen festgelegt.
 - ➔ Es gibt folgende Sichtbarkeitsmodifikatoren, welche von Klassen, Attributen und Methoden angewendet werden können:

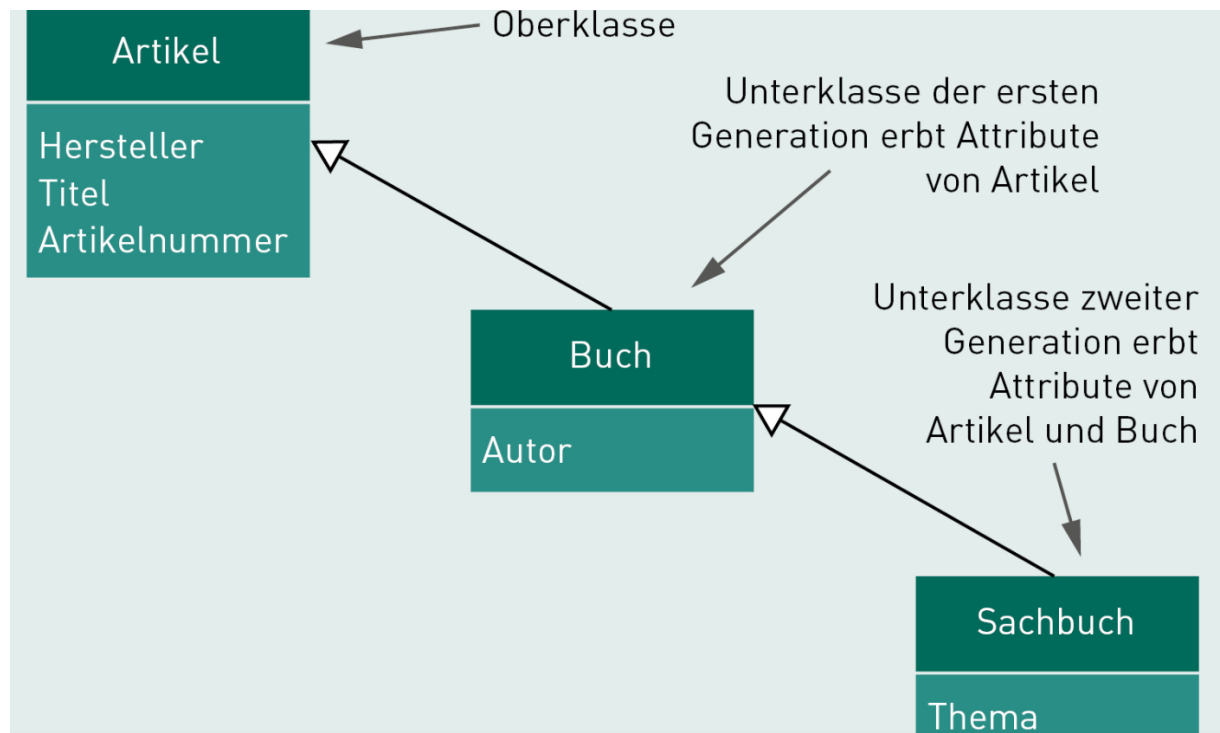
Sichtbarkeitsmodifikator	Beschreibung	Anwendbar auf	Beispiel
public	Das Element ist für alle Klassen des Programms sichtbar.	Klassen, Attribute, Methoden	<pre>public class Bestellung {} public void berechneSumme() {}</pre>
(nichts)	Das Element ist nur im gleichen Paket sichtbar.	Klassen, Attribute, Methoden	<pre>class Bestellung {} void berechne Summe() {}</pre>
protected	Das Element ist nur im gleichen Paket oder abgeleiteten Klassen (Details siehe Lektion 5) sichtbar.	Attribute, Methoden	<pre>protected void berechneSumme() {} protected int anzahlArtikel;</pre>
private	Das Element ist nur für Elemente der gleichen Klasse sichtbar.	Attribute, Methoden	<pre>private void berechneSumme() {} private int anzahlArtikel;</pre>

➔ Attribute sollten in der Regel mit dem Modifikator private deklariert werden

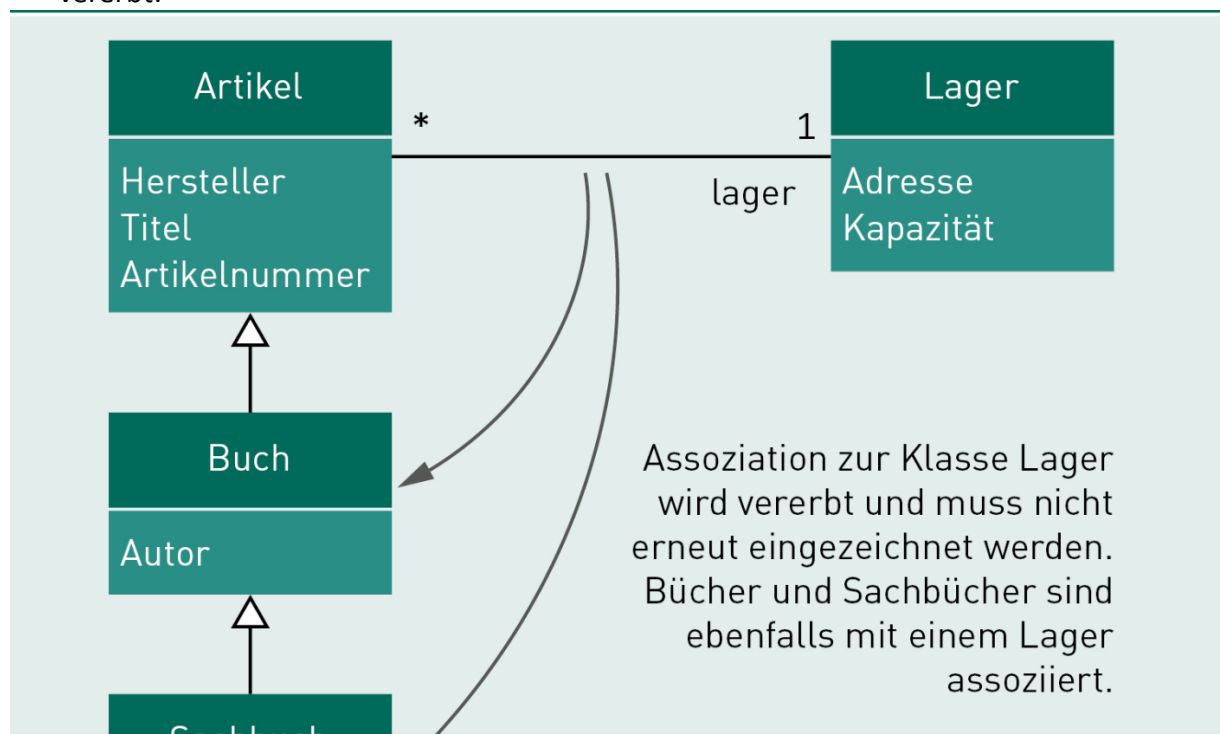
5 Vererbung

5.1 Modellierung von Vererbung im Klassendiagramm

- Attribute, die alle Klassen gemein haben, können in einer Oberklasse zusammengefasst werden und ihre Attribute an die Unterklassen vererben.
- Im UML-Klassendiagramm kennzeichnet man die „Ist-ein“ Beziehung mit einem Pfeil mit geschlossener Spitze, die nicht ausgefüllt ist.
- Die „ist-ein“ Beziehung drückt aus, dass eine Klasse eine spezielle Art einer anderen Klasse ist.
- Unterklassen sind eine spezielle Art der Oberklasse und haben alle Attribute und Methoden der Oberklasse und noch weitere nur diese Klasse betreffende Attribute
- Vererbung ermöglicht das Zusammenfassen von Gemeinsamkeiten und reduziert unnötige Wiederholungen.
- Die Vererbungsbeziehung ist transitiv, d.h. die Oberklasse vererbt alle Attribute und Methoden an die Unterklasse und an deren weitere Unterklassen. z.B. Ein Sachbuch erbt alle Attribute und Methoden von Artikel(Oberklasse) und Buch(Unterklasse)



- Es werden auch vorhandene Assoziationen der Oberklassen zu anderen Klassen vererbt:



- Es ist keine Mehrfachvererbung erlaubt. Eine Unterklasse kann nur von einer Oberklasse erben.

5.2 Programmieren von Vererbung in Java

- In Java spricht man bei der Vererbung von Erweiterung, deswegen ist das Schlüsselwort auch extends
 ➔ Das Schlüsselwort wird schon bei der Deklaration der Klasse genutzt
- Attribute und Methoden die in der Oberklasse mit private deklariert wurden, werden nicht vererbt.
- Vererbungsbeziehung geht von der abgeleiteten Klasse aus:



- Man kann auch eine Variable von Typ Artikel deklarieren und ein Objekt Buch zuweisen -> Zuweisungskompatibilität(erlaubt es einer Variablen von Typ einer Oberklasse ein Unterklasse-Objekt zuzuweisen)
 ➔ Der Typ der Variablen entscheidet, auf welche Attribute und Methoden zugegriffen werden kann.
- Zuweisungskompatibilität wird in großen Softwaresystemen verwendet um eine lose Kopplung der Klassen untereinander zu gewährleisten.
- Unterklassen erben die Attribute und Methoden sind aber nicht an die Implementierung gebunden
- Überschreiben = die erneute Implementierung einer geerbten Methode
 ➔ Methoden können in Unterklassen überschrieben werden, indem eine Methode mit der selben Signatur erstellt wird und der Methodenrumpf ergänzt wird:

```

public class Buch extends Artikel {
    ...
    public String getBeschreibung() {
        return artikelnummer + ":"
            + hersteller + ";"
            + titel
            + " von " + autor;
    }
}
  
```

Erneute Deklaration und Implementierung einer geerbten Methode

Berücksichtigung des neuen Attributs autor in der Zusammensetzung der Beschreibung


- ➔ Die überschriebene Methode können von Instanzen ausgewählt werden
- Es kann aber auch auf die in der Oberklasse deklarierte Methode zugegriffen werden mit dem Schlüsselwort super -> super geht eine Ebene in der Vererbungshierarchie hoch
 ➔ Mithilfe von super können vorhandene Implementierungen erweitert werden.

6 Wichtige objektorientierte Konzepte

6.1 Abstrakte Klassen

- Ist eine Klasse von der keine Instanzen(Objekte) erzeugt werden. -> sie fasst Gemeinsamkeiten zusammen und gibt Schnittstelle vor die von den Unterklassen erfüllt werden müssen.
 - Im UML-Diagramm werden sie durch das Schlüsselwort in geschweiften Klammern gekennzeichnet {abstract}/oder der Klassenname wird kursiv gesetzt
 - Bei der Implementierung wird das Schlüsselwort wie folgt verwendet:
-

Schlüsselwort zur Kennzeichnung von abstrakten Klassen




```
public abstract class Artikel {  
    ...  
}
```

e hat Herr Koch erreicht, dass in seinem System keine Artikel-Objekte mehr erzeugt werden dürfen.

➔ Ansonsten Verhalten sich abstrakte Klassen wie normale Klassen

- Es können nicht nur Klassen sondern auch Methoden als abstrakt gekennzeichnet sein.
 - ➔ Eine abstrakte Methode hat eine Signatur, aber keine Implementierung(Methodenrumpf)
- Definiert eine Klasse mindestens eine abstrakte Methode, so muss die Klasse ebenfalls als abstrakt gekennzeichnet sein -> Die abstrakte Methode muss zwingend in einer Unterklasse durch überschreiben implementiert werden

Klasse muss abstrakt sein, da sie eine abstrakte Methode deklariert



```
public abstract class Artikel {  
    ...  
    public abstract String getTwitterBeschreibung();  
}
```



Kennzeichnung der
abstrakten Methode



Abstrakte Methoden
definieren keinen
Methodenrumpf

- Die abstrakte Methode muss dann in einer Unterklasse implementiert werden. Das funktioniert genauso wie das überschreiben einer geerbten Methode:

```

public class Buch extends Artikel {
    ...
    public String getTwitterBeschreibung() {
        return "Buch: '" + titel + "' von " + autor;
    }
}

```

Implementierung der
abstrakten Methode in
der Unterklasse Buch



- Wenn man die abstrakte Methode nicht in der Unterklasse implementieren möchte gibt es nur eine Möglichkeit, man muss die implementierte Methode in der Unterklasse ebenfalls als abstrakt deklarieren, jedoch muss dann die Klasse auch abstract sein und die Implementierung wird auf weitere Unterklassen delegiert

6.2 Polymorphie

- Bedeutet Vielgestalt
 - ➔ Bezeichnet in objektorientierten Systemen die Tatsache, dass eine Variable, die vom type einer bestimmten Klasse deklariert ist, auch Instanzen der Unterklassen annehmen kann
 - ➔ Man kann einer Variablen Objekte unterschiedlicher Klassen zuweisen, wenn die Klassen in einer Vererbungsbeziehung zueinander stehen -> ermöglicht es mit nur einer Variablen verschiedene Methodenimplementierungen aufzurufen, je nachdem welches Objekt der Variable zugewiesen ist.
- Polymorphie ermöglicht verschiedene Arten von Artikeln im System über eine gemeinsame Schnittstelle anzusprechen und gleichzeitig auf deren spezielle Implementierung zuzugreifen

6.3 Statische Attribute und Methoden

- Klassenvariablen(auch statische Attribute) sind Attribute, die für alle Instanzen einer Klasse gleich sind und über den Klassennamen angesprochen werden.
 - ➔ Es muss kein Objekt erzeugt werden
- Sie werden mit dem Schlüsselwort static deklariert

```
public abstract class Artikel {
    public static String TRENNZEICHEN = ";";
    [...]
}
```

← Deklaration eines statischen Attributs

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Artikel.TRENNZEICHEN);
    }
}
```

← Zugriff auf das statische Attribut über den Klassennamen

- Wenn man den Wert des statischen Attributs ändert, ändert er sich für alle Objekte der Klasse und der Unterklassen
- Es gibt auch Methoden die mit static deklariert werden -> ebenfalls unabhängig von einem Objekt und werden über den Klassennamen aufgerufen:

```
public abstract class Artikel {
    public static boolean istArtikelnummerGueltig(String artikelnummer) {
        ...
    }
    ...
}
```

← Deklaration einer statischen Methode

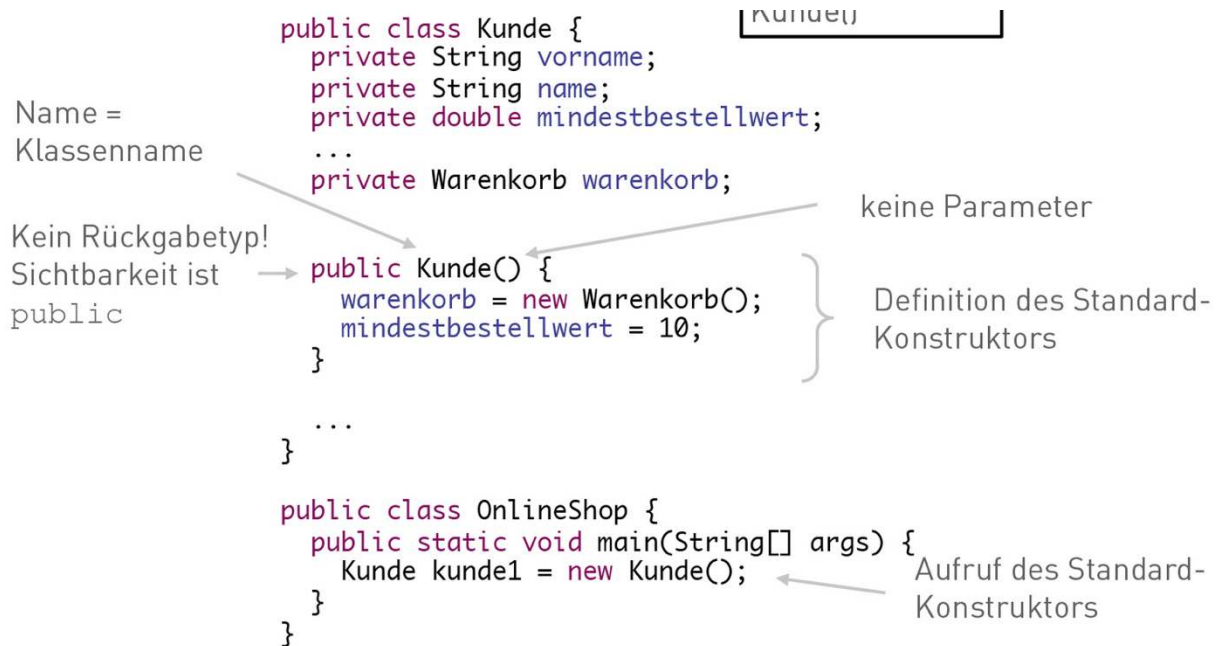
```
public class Main {
    public static void main(String[] args) {
        System.out.println(
            Artikel.istArtikelnummerGueltig("abd:1271813186317"));
    }
}
```

← Aufruf der statischen Methode Attribut über den Klassennamen

7 Konstruktoren zur Erzeugung von Objekten

7.1 Der Standard Konstruktor

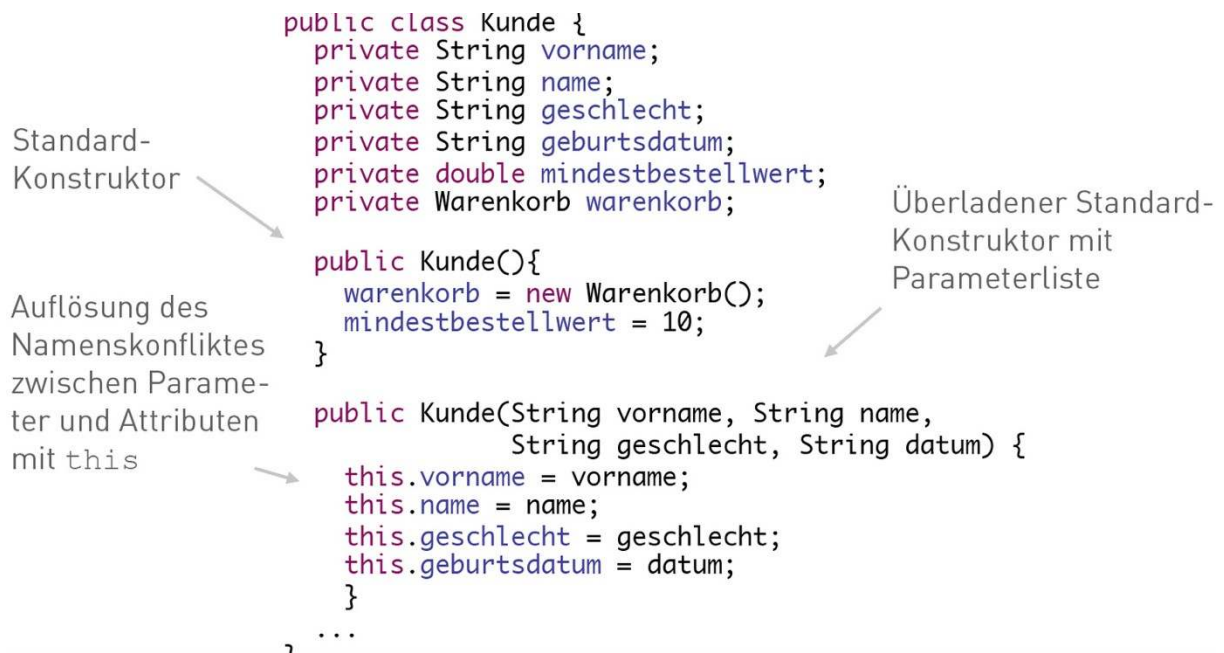
- Der new Operator erzeugt ein Objekt zu einer gegebenen Klasse -> zu dem Zweck wird er Konstruktor aufgerufen
- Der Standard Konstruktor ist eine spezielle Art einer Methode zur Erzeugung von Objekten einer Klasse. -> darf es nur einmal pro Klasse geben
- Der Standard Konstruktor ist der Syntax einer normalen Methode sehr ähnlich



- Für die Definitionen des Standard Konstruktors gelten folgende Regeln:
 - ➔ Der Name muss mit dem Klassennamen übereinstimmen
 - ➔ Er besitzt keinen Rückgabetypp
 - ➔ Sichtbarkeit sollte immer public sein-> damit andere Klassen ein Objekt dieser Klasse mit dem new-Operator erstellen können
 - ➔ Hat keine Parameter
- Zwei Aufgaben des Standard Konstruktors:
 1. Anpassen der Defaultwerte für primitive Datentypen
 2. Erzeugung von nicht-primitiven Attributen (Objekten)
- Wenn die beiden Aufgaben für eine Klasse nicht notwendig sind, kann der Standard Konstruktor weggelassen werden -> er wird vom Java-Compiler automatisch erstellt
 - ➔ Er ist im Quelltext nicht sichtbar es kann aber mit dem new-Operator ein Objekt erzeugt werden.
- Bei der Verwendung von Konstruktoren ist zu beachten, dass nicht mehr benötigte Objekte vom der Java-DIE gelöscht werden -> Garbage Collection/automatische Speicherverwaltung
 - ➔ Bei Bedarf kann der Garbage Collector durch die Anweisung `System.gc()` aufgerufen werden

7.2 Überladen von Konstruktoren

- Es ist üblich Objekte direkt nach der Erstellung mit Werten zu befüllen
- Um Redundanzen zu vermeiden, kann der Standard Konstruktor überladen werden -> d.h. dass ein zweiter Konstruktor erstellt wird, mit einer Parameterliste mit den Attributen, die gefüllt werden müssen:



- Mit Schlüsselwort this wird der Namenskonflikt zwischen Parametern und Attributen aufgehoben

Aufruf des überladenen Konstruktors:

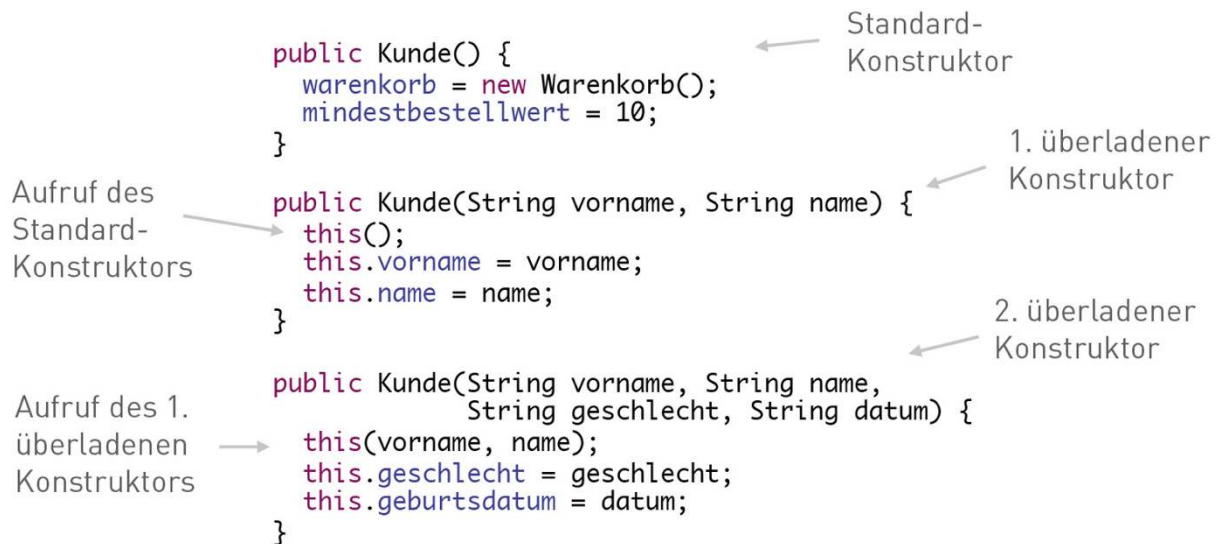
```

public Kunde neuerKunde(String vorname, String name, String geschlecht,
                        String datum) {
    Kunde k = new Kunde(vorname, name, geschlecht, datum);
    return k;
}

```

Aufruf des überladenen Konstruktors

- Wenn jedoch andere Attribute bei der Erstellung eines neuen Kunden benötigt werden, kann ein weiterer überladener Konstruktor erstellt werden. -> es können beliebig viele Konstruktoren erstellt werden, solange sich die Parameterlisten unterscheiden.
- Bei der Definition mehrere überladener Konstruktoren, ruft man den ersten überladenen Konstruktor mit this() auf und fügt weitere Attribute hinzu:



- Die Abarbeitungsreihenfolge: zuerst wird der Standard-Konstruktor aufgerufen, mit werten befüllt und dann ruft der Standard Konstruktor den 1. Überladenen Konstruktor auf
- Man kann mit Konstruktoren auch Sicherheitskopien eines vorhandenen Objektes erstellen:

▼ Code

```
Kunde neuesKundenObjekt = new Kunde(bestehendesKundenObjekt);
```

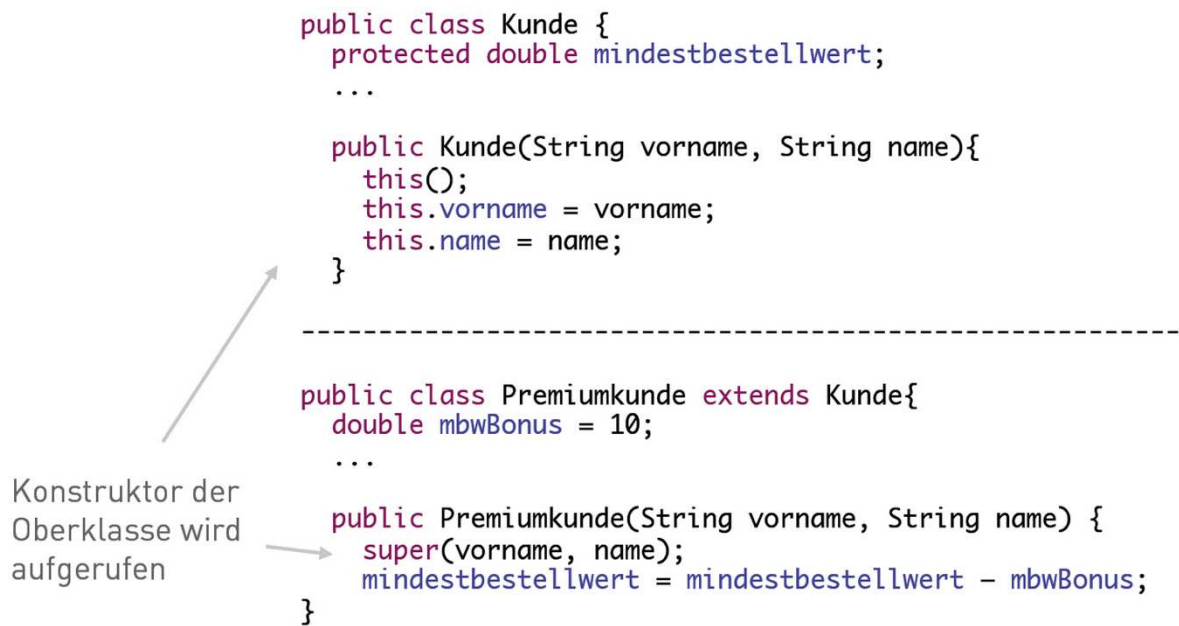
Abbildung 76: Copy-Konstruktor, der eine Kopie eines Kunden-Objektes erzeugen kann

```

public Kunde(Kunde original){
    vorname = original.vorname;
    name = original.name;
    geschlecht = original.geschlecht;
    geburtsdatum = original.geburtsdatum;
    warenkorb = original.warenkorb;
    mindestbestellwert = original.mindestbestellwert;
}

```

- Flache Kopie -> nur das Objekt selbst wird geklont, jedoch nicht die referenzierten Objekte
- Tiefe Kopie -> wenn das referenzierte Objekt nicht direkt sondern ebenfalls über einen Copy Konstruktor aufgerufen wird.
- Es kann bei Vererbungshierarchien auch der Konstruktor der Superklasse aufgerufen werden mit dem Schlüsselwort super:



8 Ausnahmebehandlung von Exceptions

8.1 Typische Szenarien der Ausnahmebehandlung

- Ausnahme = ein Zustand, der das Programm an der Fortführung des normalen Ablaufs hindert
- Bei einer falschen Eingabe sollen durch Konzepte zur Ausnahmebehandlung vermieden werden, dass das System abstürzt.
- Fehlersignal: ein solches kodiert eine Ausnahme anhand eines Wertes der außerhalb des fachlich gültigen Wertebereichs liegt -> das Programm muss die Kodierung kennen um auf die Ausnahme reagieren zu können

Behandlung von Fehlersituationen mit Signalen:



```

-
public class OnlineShop {
    ...
    public void zeigeStatistik (Warenkorb w) {
        double ppa = w.preisProArtikel();
        ...
        if (ppa == -1) ←
            System.out.println ("Ungültiger Preis pro Artikel.");
        else
            System.out.println ("Der Preis pro Artikel lautet:" + ppa);
    }
}

```

- Großer Nachteil in der Praxis, da Funktionen nur einen Rückgabewert haben und Fehlersignale müssen über den Rückgabewert an das Programm gemeldet werden -> führt zu schlechter Softwarequalität
- Für robustere Programme sollte es einen eigenen Kanal geben für die Signalisierung von Fehlern geben, die zur Entkopplung von Fehlersignalen und Rückgabewerten führen
Weiterer Vorteil so einer Datenstruktur: differenzierte Fehlersignale definieren zu können
Zudem wäre es wünschenswert mögliche Ausnahmen einer Methode abfangen zu müssen damit sichergestellt wird, dass eine Lösungsstrategie zu häufigen Fehlerquellen existieren

8.2 Standard Exceptions in Java

- Unchecked Exceptions = alle Exceptions die von der Klasse RuntimeException abgeleitet werden
- Exception ist ein objektorientiertes Konzept zur Definition und Verwendung von Ausnahmen in Java.
Es gibt viele vordefinierte Standard Exceptions z.B.:

ArithmeticException	Division durch Null	int n = 0; return 1/n;
ArrayIndexOutOfBoundsException	Zugriff auf ein Listenelement außerhalb des definierten Bereichs (hierzu später mehr)	int[] zahlen = new int [8]; zahlen [10] = 4;
NullPointerException	Zugriff auf ein nicht instanziiertes Objekt	kunde k = null; k.setName("Meier");

- Beim Auftreten von Fehlern bietet Java mehrere Alternative vorgehensweisen:
1. Die Exception wird innerhalb der Methode abgefangen

Ausnahmen werden mit einem try/catch-Block abgefangen. Im try-Block stehen die kritischen Programmanweisungen. Im catch-Block werden die Ausnahmen behandelt:

Ausnahmen im try-Block können im catch-Block behandelt werden

```
public class Warenkorb {

    private int anzahlArtikel;
    private float artikelSumme;
    ...

    public double preisProArtikel(){
        double ergebnis;
        try {
            ergebnis = artikelSumme / anzahlArtikel;
        }
        catch (ArithmeticException ex) {
            ergebnis = 0;
        }
        return ergebnis;
    }
}
```

Angabe, welche Exception abgefangen werden kann

2. Die Exception wird an das aufrufende Programm weitergeleitet
Damit Methoden Exceptions weiterleiten können muss ihre Signatur um das Schlüsselwort throws ergänzt werden. -> es folgt eine Auflistung aller Exceptions die in der Methode auftreten können-> aufrufende Programme erkennen welche Ausnahmen von der Methode zu erwarten sind. -> der try/catch-Block wird in das aufrufende Programm verlagert.

```
public class Warenkorb {
    ...
    public double preisProArtikel() throws ArithmeticException{
        return artikelSumme / anzahlArtikel;
    }
}
```

Gibt aufrufendem Programm bekannt, welche Exceptions hier auftreten können

```
public class OnlineShop {
    public void zeigeStatistik(Warenkorb w) {

        try {
            double ppa = w.preisProArtikel();
            System.out.println(ppa);
        }
        catch (ArithmeticException ex) {
            System.out.println(ex.getMessage());
        }
        ...
    }
}
```

Das aufrufende Programm muss per try/catch-Block für die Ausnahmebehandlung sorgen ...

... sonst bricht das Programm ab.

Es gibt im Ausnahmefall die Fehlermeldung aus.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ausnahmebehandlung.Warenkorb.preisProArtikel(Warenkorb.java:76)
    at ausnahmebehandlung.OnlineShop.main(OnlineShop.java:13)
```

- Fehlermeldung ist für User unverständlich, da nicht die Ursache genannt wird.

3. Die Exception wird abgefangen und dann mit einer spezifischen Meldung an das aufrufende Programm weitergeleitet
= kontextbezogene Fehlermeldung, die auf das tatsächliche Problem hinweist.
Hier wird die exception ebenfalls weitergeleitet und mit dem Schlüsselwort throw eine spezifische Fehlermeldung ausgegeben.

Es wird in der Methode die Exception wieder im try/catch-Block gefangen und dann eine neue `ArithmeticException` zu werfen, die die Fehlermeldung enthält. Die Fehlermeldung wird dem Konstruktor der Exception in den Parametern übergeben:

```
public class Warenkorb {
    ...

    public double preisProArtikel() throws ArithmeticException {
        double ergebnis;
        try {
            ergebnis = artikelSumme / anzahlArtikel;
        }
        catch (ArithmeticException ex) {
            throw new ArithmeticException("Berechnung unmöglich: " +
                "Warenkorb ist leer!");
        }
        return ergebnis;
    }
}
```

Die Ausnahme wird gefangen und erneut geworfen, um eine spezifische Fehlermeldung einzubauen.

- Es können zu einem try Block auch mehrere catch-Blöcke definiert werden. Das macht Sinn wenn:
 1. Im try-Block viele unterschiedliche Exceptions auftreten können und jede dieser Exceptions separat behandelt werden sollen
 2. Sichergestellt werden soll, dass neben einer bestimmten Standard-Exception auch alle anderen Exceptions abgefangen werden sollen

```
public class Warenkorb {
    ...
    public double preisProArtikel()
        throws ArithmeticException, ArrayIndexOutOfBoundsException {
        ...
    }
}
```

Eine weitere Exception kann geworfen werden.

```
public class OnlineShop {
    public void zeigeStatistik(Warenkorb w) {

        try {
            double ppa = w.preisProArtikel();
            System.out.println(ppa);
        }
        catch (ArithmeticException ex) {
        }
        catch (ArrayIndexOutOfBoundsException ex) {
        }
        catch (Exception ex) {
        }
        ...
    }
}
```

Nur ausgewählte Exceptions werden gefangen; hier also spezielle Fehler behandeln

Alle übrigen Exceptions werden abgefangen; hier also alle allgemeinen Fehler behandeln

- Finally-Block: hier werden alle Anweisungen festgehalten unabhängig, ob eine Ausnahme aufgetreten ist oder nicht -> meist werden Aufräumarbeiten erledigt:

```

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

    catch (Exception ex) {
    }
    finally {
    }
}

```

← Allgemeine Fehlerbehandlung

← hier Aufräumarbeiten erledigen, die unabhängig vom Auftreten einer Exception ausgeführt werden müssen

8.3 Definieren eigener Exceptions

- Eigene Exceptions : können Programmspezifische Ausnahmen abbilden -> müssen von der Klasse Exception abgeleitet werden

Eigene Ausnahmen müssen – wie Standard-Exceptions auch – von Exception erben.

```

public class MindestbestellwertNegativException extends Exception
{
    public MindestbestellwertNegativException()
    {
        super("Der Mindestbestellwert ist negativ!");
    }

    public MindestbestellwertNegativException(String message)
    {
        super(message);
    }
}

```

Beim Aufruf des Standard-Konstruktors soll eine vordefinierte Meldung ausgegeben werden.

↑
Wird dem Konstruktor eine Zeichenkette übergeben, wird diese als Nachricht interpretiert und an den Oberklassen-Konstruktor weitergeleitet.

- Jede Exception, ob eigene oder standard muss von der Oberklasse Exception erben.
- Mithilfe der Konstruktoren, kann festgelegt werden, welche Nachrichten von der Exception erzeugt werden.

Werfen einer eigenen Exception:

```

public class Kunde {
    ...
    public double getMindestbestellwert() throws
    MindestbestellwertNegativException
    {
        if (mindestbestellwert >= 0)
            return mindestbestellwert;
        else
            throw new MindestbestellwertNegativException();
    }
}

```

← Ist der Wert negativ, wird die eigene Exception mit der oben definierten Standard-Nachricht erzeugt.

Fangen einer eigenen Exception:


```

public class OnlineShop {
    ...
    public Kunde getKundendaten(int kundenNr) {

        Kunde k = kundenliste.get(kundenNr);
        double mbw = 0;

        try {
            mbw = k.getMindestbestellwert();
        }
        catch (MindestbestellwertNegativException ex){
            System.out.println(ex.getMessage());
        }
        ...
    }
}

```

Anstelle der Überprüfung des Rückgabewertes auf enthaltene Fehlersignale wird nun die eigene Exception gefangen und behandelt.

9 Programmierschnittstellen mit Interfaces

9.1 Typische Szenarien für Programmierschnittstellen

- Gang of Four = eine Gruppe von vier Autoren, die Literatur zum Thema Softwareentwicklung verfasst
 - ➔ Es sei ein besserer Programmierstil gegen ein Interface zu programmieren als gegeben eine Implementierung -> sorgt für eine klare Trennung von Spezifikation(was sollen Klassen oder Pakete können) und Implementierung(wie wird die Funktionalität umgesetzt)

Zwei Vorteile bei dieser Trennung:

1. Flexiblere Softwarearchitektur da die Implementierung bei Bedarf ausgetauscht werden kann ohne viel Aufwand
2. Erhöht die Wiederverwendung von Klassen und Paketen -> eine Klasse die eine bestimmte Funktionalität anbietet, kann überall dort implementiert werden, wo diese Funktionalität gebraucht wird.

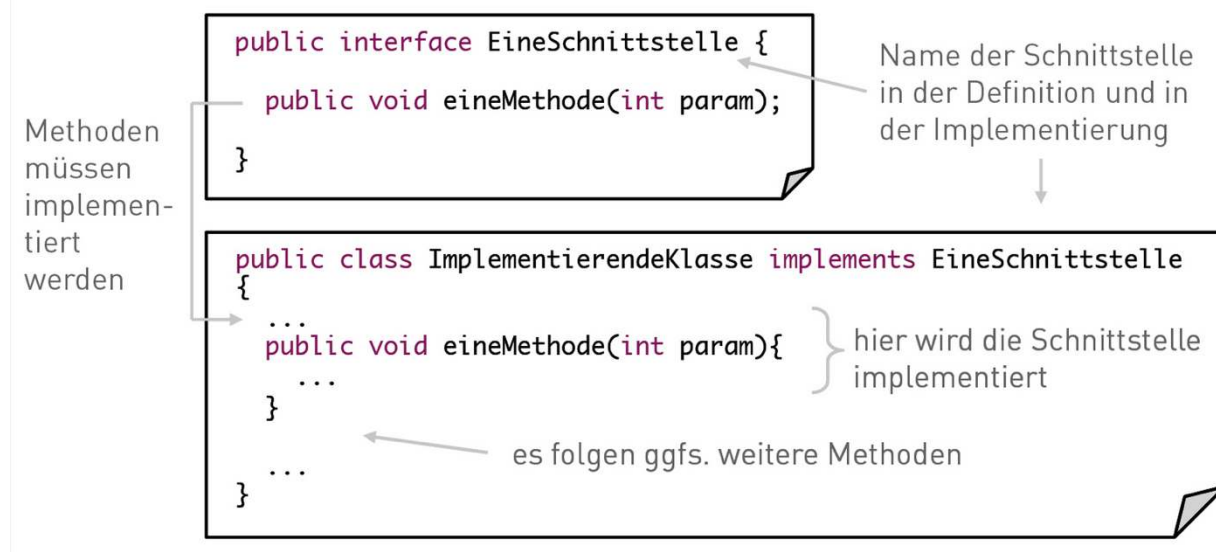
Interfaces eignen sich als Instrument für Polymorphie -> bei Methoden können die Methoden der Oberklassen von Unterklassen unterschiedliche neu implementiert werden.

- ➔ Wenn alle Klassen der Vererbungshierarchie in eine gemeinsame Liste gespeichert werden und die Methode der Oberklasse aufgerufen wird, wird jeweils die passende Implementierung von der DIE ausgewählt = dynamisches Binden
- ➔ Mit Interfaces kann die Einschränkung aufgehoben werden, dass sich die Klassen in einer Vererbungshierarchie befinden müssen -> entscheidend ist nur, dass sie ein gemeinsames Interface implementieren

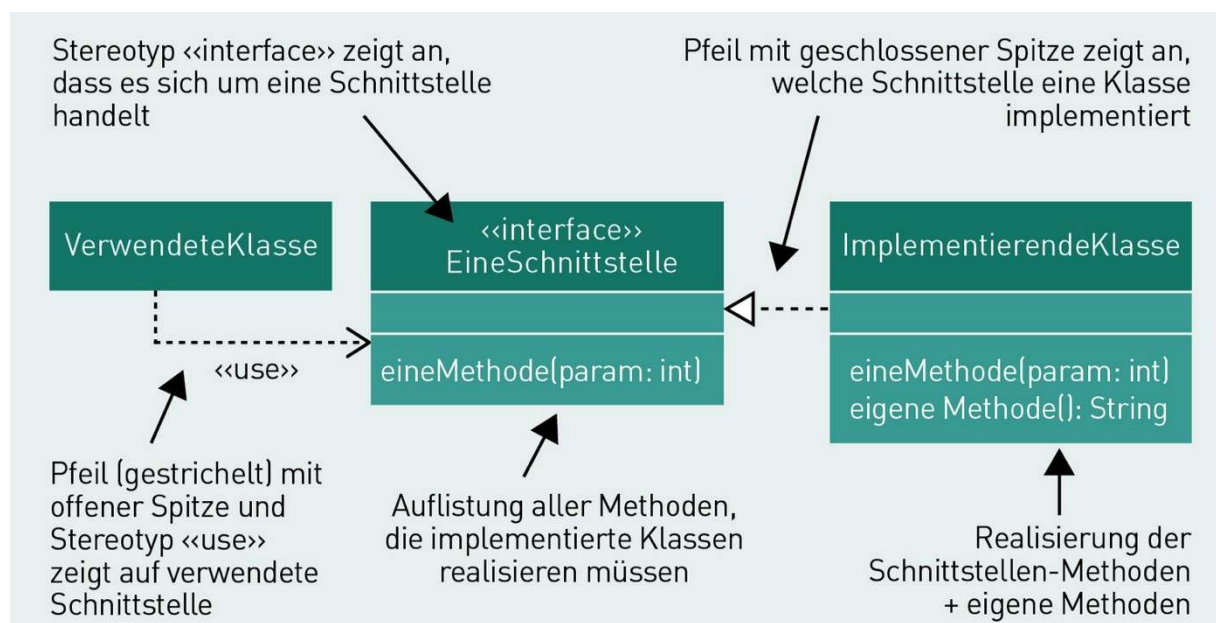
9.2 Interfaces als Programmierschnittstellen in Java

- Schnittstellen werden in Java mit dem Schlüsselwort interface definiert
- Syntax ist ähnlich dem einer Klasse, nur dass alle Methoden abstract sind -> Methoden bestehen nur aus Signaturen -> so wird definiert, was eine Klasse können muss, ohne das Wie vorwegzunehmen.
- Es dürfen keine Attribute definiert werden nur mit Konstanten

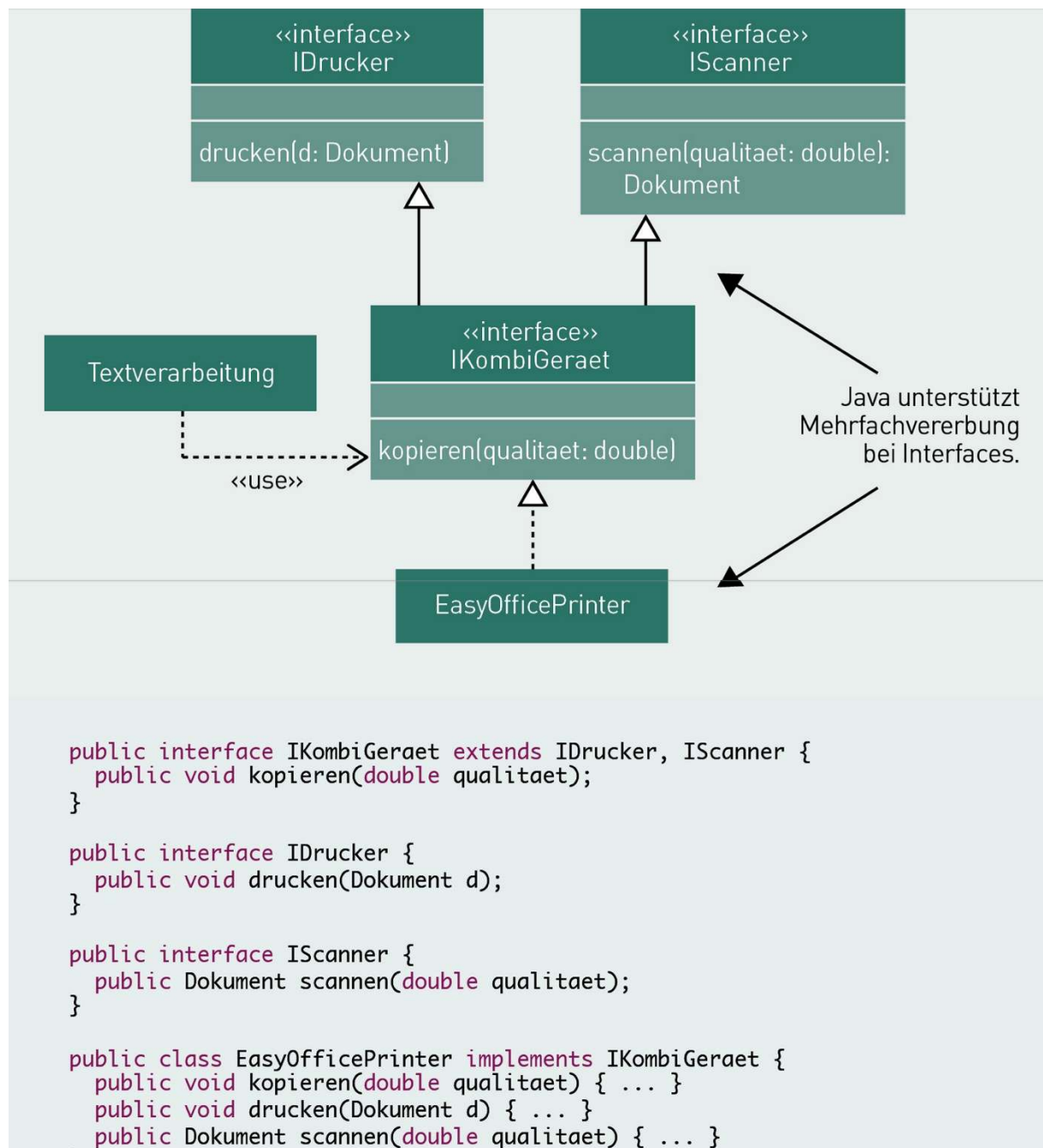
Definition und Implementierung einer Schnittstelle:



- Mit Schlüsselwort `implements` kann eine Klasse zeigen, welche Schnittstelle sie realisiert -> jede Methode der Schnittstelle muss implementiert werden
- Im UML_Diagramm ist die Notation der Schnittstelle ähnlich den der Klassen -> es wird mit Stereotyp `<<interface>>` und einer use-Assoziation dargestellt



- Durch Verwendung von Interfaces können benötigte Methoden beliebig ausgetauscht werden -> Flexibilität und an anderen Stellen im Programm eingesetzt werden, wo so ein Interface benötigt wird -> Wiederverwendbarkeit
- Interfaces können von anderen Interfaces erben -> so können Gemeinsamkeiten zusammengefasst werden und ohne großen Wartungsaufwand erweitert werden
- Mehrfachvererbung bei Interfaces -> Interfaces können von mehr als einem Interface erben im Gegensatz zu Klassen die nur von jeweils einer Klasse erben können. Trotzdem Schlüsselwort `extends`.



- Unterschiede zu abstrakten Klassen -> die Vererbungshierarchie spielt keine Rolle
- Gemeinsamkeiten zu abstrakten Klassen -> es können Funktionalitäten vorgeschrieben werden