JUSTUS-LIEBIG-UNIVERSITÄT GIESSEN
PROFESSUR BWL – WIRTSCHAFTSINFORMATIK
UNIV.-PROF. DR. AXEL SCHWICKERT

Schwickert, Axel C.; Patzak, Maximilian; Schick; Lukas;
Schramm, Laura; Kuchenbrod, Ramona

Prgrammierung mit Java – Teil 2 – Reader zur WBT-Serie

ARBEITSPAPIERE WIRTSCHAFTSINFORMATIK

Nr. 01/2017
ISSN 1613-6667

Arbeitspapiere WI Nr. 1 / 2017

Autoren: Schwickert, Axel C.; Patzak, Maximilian; Schick; Lukas; Schramm, Laura; Kuchenbrod, Ramona

Titel: Programmierung mit Java – Teil 2 – Reader zur WBT-Serie

Zitation: Schwickert, Axel C.; Patzak, Maximilian; Schick; Lukas; Schramm, Laura; Kuchenbrod, Ramona: Programmierung mit Java – Teil 2 – Reader zur WBT-Serie, in: Arbeitspapiere WI, Nr. 1/2017, Hrsg.: Professur BWL – Wirtschaftsinformatik, Justus-Liebig-Universität Gießen 2017, 100 Seiten, ISSN 1613-6667.

Kurzfassung: Das vorliegende Arbeitspapier dient als Reader zur WBT-Serie „Programmierung mit Java – Teil 2“, die im E-Campus Wirtschaftsinformatik online zur Verfügung steht.

Zunächst wird die integrierte Entwicklungsumgebung „Eclipse“ vorgestellt und eingeführt. Darauf folgend werden mithilfe von Eclipse erste Objekte erzeugt. Diese ersten Objekte stellen Studierende und Professoren dar und werden genutzt, um die Vererbung in Java zu demonstrieren. Abstrakte Klassen und das Überschreiben von Methoden wird anschließend am Beispiel von Kraftfahrzeugen erläutert. Das Konzept der Polymorphie beschreibt weiterhin, wie ein Bezeichner abhängig von seiner Verwendung Objekte unterschiedlichen Datentyps annehmen kann. Die Überladung bezeichnet das bewusste Einführen dieser Polymorphien. Mithilfe des Konezepts der Verkapselung wird anschließend demonstriert, wie die Bespiele Autos und LKW ihre Eigenschaften verbergen können. Um die Beispiele Autos und LKW in einem Vermietungsunternehmen "mietbar" machen zu können, wird auf Interfaces zurück gegriffen. Abschließend erfolgt eine Zusammenfassung der unterschiedlichen Konzepte anhand eines Webshop-Beispiels.

Schlüsselwörter: Programmierung mit Java, Unified Modeling Language (UML), Objektorientierte Programmierung

A Die Web-Based-Trainings

Der Stoff zum Thema „Programmierung mit Java – Teil 2“ wird durch eine Serie von Web-Based-Trainings (WBT) vermittelt. Die WBT bauen inhaltlich aufeinander auf und sollten daher in der angegebenen Reihenfolge absolviert werden. Um einen Themenbereich vollständig durchdringen zu können, muss jedes WBT mehrfach absolviert werden, bis die jeweiligen Tests in den einzelnen WBT sicher bestanden werden.

WBT-Nr.	WBT-Bezeichnung	Bearbeitungs-dauer
1	Einführung in die Entwicklungsumgebung Eclipse	90 Min.
2	Erste Objekte mit Eclipse erzeugen	60 Min.
3	Vererbung am Beispiel von Studierenden und Professoren	45 Min.
4	Abstraktion und Überschreibung am Beispiel von Kraftfahrzeugen	60 Min.
5	Überladung und Polymorphie am Beispiel von Autos und LKW	45 Min.
6	Autos und LKW verbergen durch Kapselung ihre Eigenschaften	45 Min.
7	Mit einem Interface Kraftfahrzeuge mietbar machen	45 Min.
8	Zusammenfassung am Beispiel des Webshops der Lemonline AG	90 Min.

Tab. 1: Übersicht WBT-Serie

Die Inhalte der einzelnen WBT werden nachfolgend in diesem Dokument gezeigt. Alle WBT stehen Ihnen rund um die Uhr online zur Verfügung. Sie können jedes WBT beliebig oft durcharbeiten. In jedem WBT sind Quellcode-Beispiele enthalten, die Sie unbedingt nachbauen und ausführen sollten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Seite

Tabellenverzeichnis

Seite

Abkürzungsverzeichnis

IDE ..... Integrated Development Environment
UML ..... Unified Modeling Language

1. Einführung in die Entwicklungsumgebung Eclipse

1.1 Praktikant/in gesucht

Anna Schmitt studiert BWL an der Justus-Liebig-Universität Gießen. Ihr Professor für Wirtschaftsinformatik hat sie in der letzten Vorlesung auf eine interessante Praktikumsstelle aufmerksam gemacht, die im Career Center des Fachbereichs Wirtschaftswissenschaften veröffentlicht wurde.

Wir, die Lemonline AG, sind nicht nur einer der führenden Hersteller von Smartphones und Tablet-PCs in Deutschland, sondern auch in der Anwendungsentwicklung tätig. Zum nächstmöglichen Zeitpunkt suchen wir einen Praktikanten in der Java-Anwendungsentwicklung (m/w) zur Verstärkung unserer IT-Abteilung in Frankfurt am Main.

Ihre Aufgaben: Sie pflegen bestehende Java-Anwendungen mit der Entwicklungumgebung Eclipse. Sie modellieren objektorientierte Java-Anwendungen auf der Basis von UML-Klassendiagrammen.

Ihre Qualifikationen: Starke analytische und konzeptionelle Fähigkeiten; engagierte und selbstständige Arbeitsweise; Grundkenntnisse in der Java-Programmierung.

Beginn: nach Absprache
Dauer: 2 – 6 Monate
Vergütung: 800 € pro Monat

Abb. 1: Stellenausschreibung der Lemonline AG

1.2 Java-Programmierung mit dem Texteditor

1.2.1 Wiederholung: Programmierung mit Java - Teil 1

Anna Schmitt: „Ich habe im letzten Semester die Veranstaltung "Programmierung mit Java - Teil 1" besucht. Die Inhalte dieser Veranstaltung fand ich sehr spannend. Da ich gerne meine Java-Kenntnisse bei einem Praktikum in der Java-Anwendungsentwicklung vertiefen möchte, werde ich mich umgehend bei der Lemonline AG bewerben. Zur Vorbereitung auf das Praktikum werde ich noch einmal einen Blick in meine Übungsunterlagen "Programmierung mit Java - Teil 1" werfen, um meine Java-Kenntnisse aufzufrischen.“

1.2.2 Wiederholung: Vorbereitungen zum Programmieren

Bevor es mit der Java-Programmierung losgehen kann, muss das Herzstück einer Java-Installation, das sogenannte Java Development Kit (JDK), installiert und die Systemvariablen anpasst werden.

Das Java Development Kit (JDK) beinhaltet den Java-Compiler und die Java Virtual Machine. Das JDK kann unter dem folgendem Link heruntergeladen werden:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Es muss sichergestellt werden, dass das JDK passend für das Betriebssystem heruntergeladen wird! Das JDK wird unter Windows auf der Festplatte C:\ installiert. Unter folgendem Link finden Sie eine Installationsanleitung:

http://docs.oracle.com/javase/8/docs/technotes/guides/install/windows_jdk_install.hml

Bitte führen Sie die folgenden Schritte durch, um die Systemvariablen anzupassen:

1. Wählen Sie im Windows-Startmenü den Menüpunkt "Computer".
2. Danach wählen Sie die Option "Systemeigenschaften" und klicken auf "Erweiterte Systemeinstellungen".
3. Im Anschluss daran klicken Sie auf "Umgebungsvariablen" und wählen unter "Systemvariablen" PATH aus. Klicken Sie auf die Option "Bearbeiten", um die vorhandenen Pfade der PATH-Systemvariable zu öffnen.
4. Überprüfen Sie, ob der folgende Pfad vorhanden ist:
   C:\Program Files\Java\jdk<Version>\bin
5. Wenn der Pfad bereits vorhanden ist, können Sie diese Seite überspringen. Wenn der Pfad noch nicht vorhanden ist, muss der Pfad *C:\Program Files\Java\jdk<Version>\bin* hinzugefügt werden. Dabei muss die *<Version>* durch die auf dem Computer installierte JDK-Version (z. B. "*C:\Program Files\Java\jdk1.8.0_25\bin*") ersetzt und die Systemvariable gespeichert werden.
6. (Alternative:) Falls die Systemvariable PATH nicht vorhanden ist, muss sie unter "Neu..." erstellt werden. Als Name wird "PATH" gewählt und danach der Pfad "*C:\Program Files\Java\jdk<Version>\bin*" als Wert hinzugefügt.

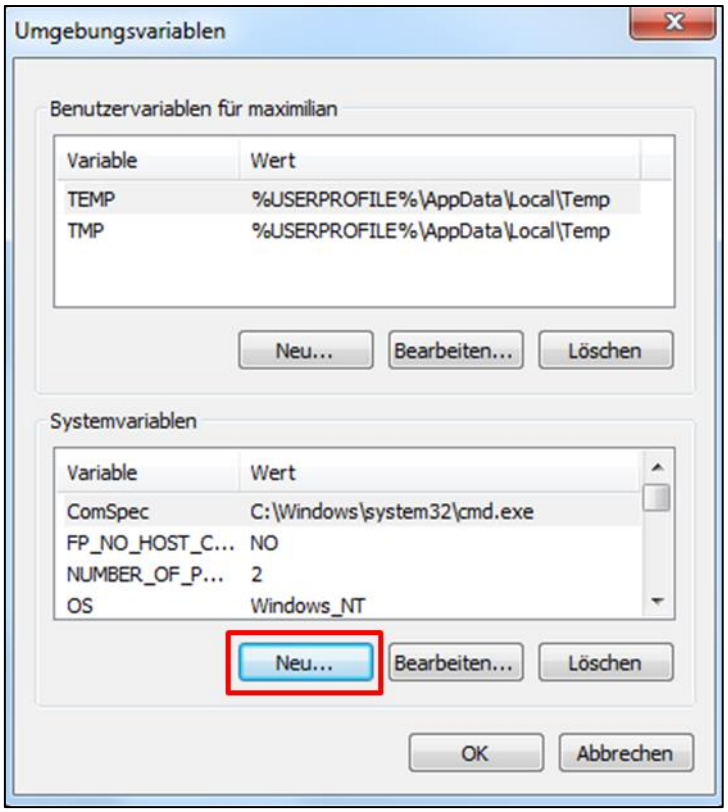


Abb. 2: Neue Umgebungsvariable anlegen

1.2.3 Wiederholung: Java-Quellcode mit dem Texteditor erstellen

Als nächstes kann es mit der Programmierung eines Java-Programms losgehen. Ich will ein Programm erstellen, dass den Satz "Hallo, Welt!" auf der Konsole ausgibt. Dafür müssen die folgenden beiden Schritte durchgeführt werden:

1. Zuerst wird der Texteditor geöffnet und der Quellcode eingegeben. Für die Programmierung des "HalloWelt"-Programms genügt ein einfacher Texteditor. Mögliche Optionen sind der "Editor" unter Microsoft Windows oder der "TextEditor" unter Mac OS.

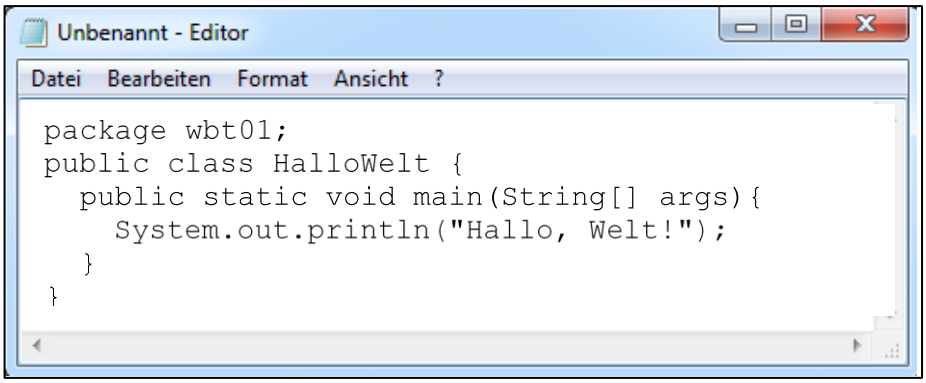


Abb. 3: Quellcode in den Texteditor eingeben

2. Danach wird der Quellcode mit dem Namen „HalloWelt.java“ gespeichert. Es ist wichtig, das Java-Programm mit der Dateiendung ".java" zu speichern. Der Java-Compiler erkennt Java-Quellcode nur dann, wenn die Datei mit ".java" endet! Hier wird der Speicherort C:\Workspace\mein_javaprojekt\wbt01 gewählt.

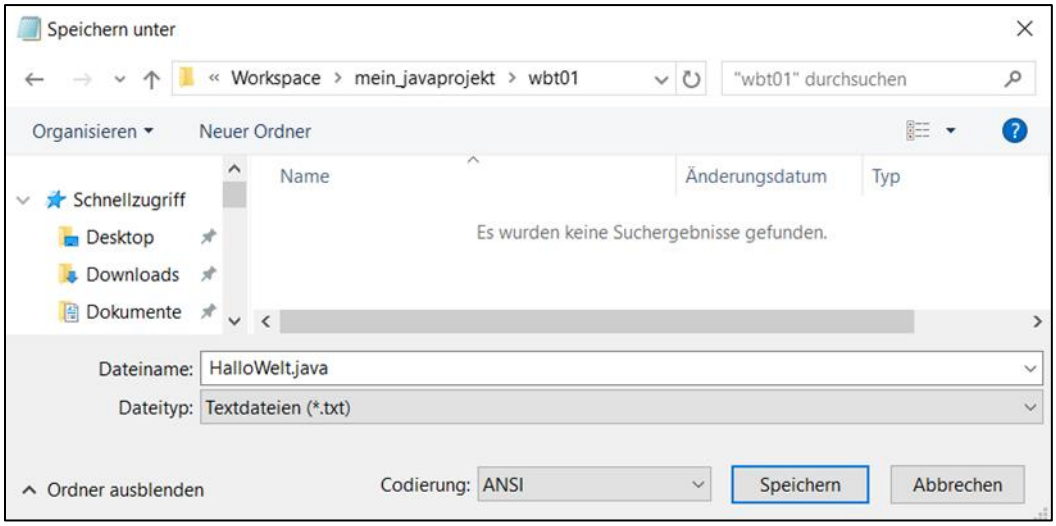


Abb. 4: Datei “HalloWelt.java“ speichern

1.2.4 Wiederholung: Java-Quellcode in der Konsole ausführen

Nachdem der Quellcode (HalloWelt.java) erstellt wurde, muss dieser mit dem Java-Compiler in Bytecode umgewandelt und mit der Java Virtual Machine ausgeführt wer-den. Dafür müssen die folgenden Befehle in die Konsole (Windows: “Eingabeaufforderung“, Mac OS: “Terminal“) eingegeben werden:

- cd C:\workspace\mein_javaprojekt\wbt01

Mit diesem Befehl wird in den Ordner mein_javaprojekt gewechselt, in dem der Quellcode für das "HalloWelt"-Programm liegt.

- javac HalloWelt.java

Dieser Befehl ruft den Java-Compiler auf und wandelt den Java-Quellcode in Bytecode um (HalloWelt.class).

- java HalloWelt

Dieser Befehl ruft die Java Virtual Machine auf und führt den Bytecode des Ja-va-Programms aus. In der Konsole erscheint der Satz "Hallo, Welt!".

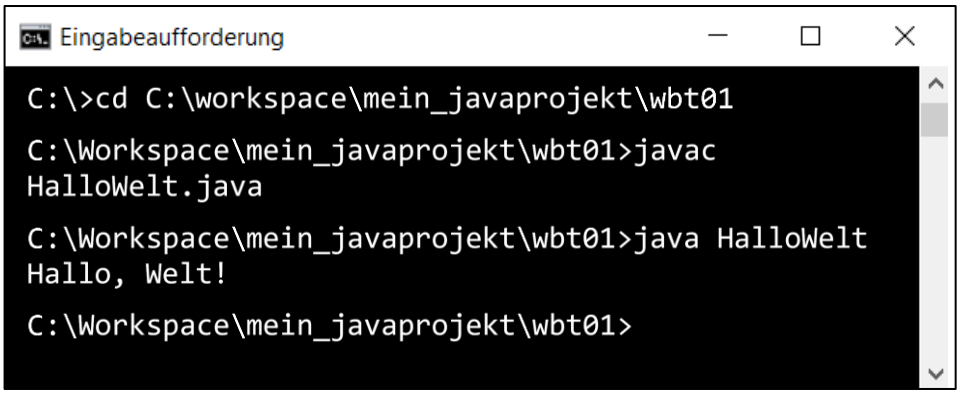


Abb. 5: “HalloWelt.java“ in der Konsole ausführen

1.2.5 Zusage für das Praktikum

Anna Schmitt: „Ich habe eine Zusage für das Praktikum in der Java-Anwendungsentwicklung erhalten und werden schon am nächsten Montag beginnen. Bei der Lemonline AG wird der Java Chef-Entwickler Horst Schäfer mein Praktikumsbetreuer sein. Nachdem ich wiederholt habe, wie man Java-Programme mit dem Texteditor erstellt und auf der Konsole ausführt, fühle ich mich gut auf das Praktikum in der Java-Anwendungsentwicklung vorbereitet.“

1.3 Die Entwicklungsumgebung Eclipse

1.3.1 Das Praktikum beginnt

Horst Schäfer: „Hallo Frau Schmitt, herzlich willkommen! Ich heiße Horst Schäfer und bin der Java Chef-Entwickler hier bei der Lemonline AG. Ich werde in den nächsten Wochen Ihr Ansprechpartner sein. In Ihrem Vorstellungsgespräch habe ich erfahren, dass Sie bisher Ihre Java-Programme mit dem Texteditor erstellt und in der Konsole ausgeführt haben. Für die Entwicklung von komplexeren Java-Programmen ist das aber viel zu umständlich. Die Gründe dafür sind unter anderem, dass:

- Sie leicht den Überblick über Ihren Quellcode verlieren,
- ein Texteditor Sie beim Identifizieren von Fehlerquellen nicht untestützt und
- das andauernde Kompilieren und Ausführen der Java-Programme auf der Konsole viel zu aufwändig ist.

Daher verwenden wir hier bei der Lemonline AG eine sogenannte "integrierte Entwicklungsumgebung".“

1.3.2 Integrierte Entwicklungsumgebung

Eine sogenannte integrierte Entwicklungsumgebung (Integrated Development Environment, kurz IDE) führt verschiedene Software-Entwicklungswerkzeuge wie den Editor und Compiler unter einer Oberfläche zusammen. Dadurch wird die Entwicklung von Java-Programmen einfacher und effizienter, weil eine IDE

- die einzelnen Bestandteile des Quellcodes farblich hervorhebt,
- die Dauer der Fehlerbehebung durch die Markierung von Fehlern im Quellcode reduziert und
- wiederkehrende Prozesse wie das Kompilieren und Ausführen von Java-Programmen automatisiert.

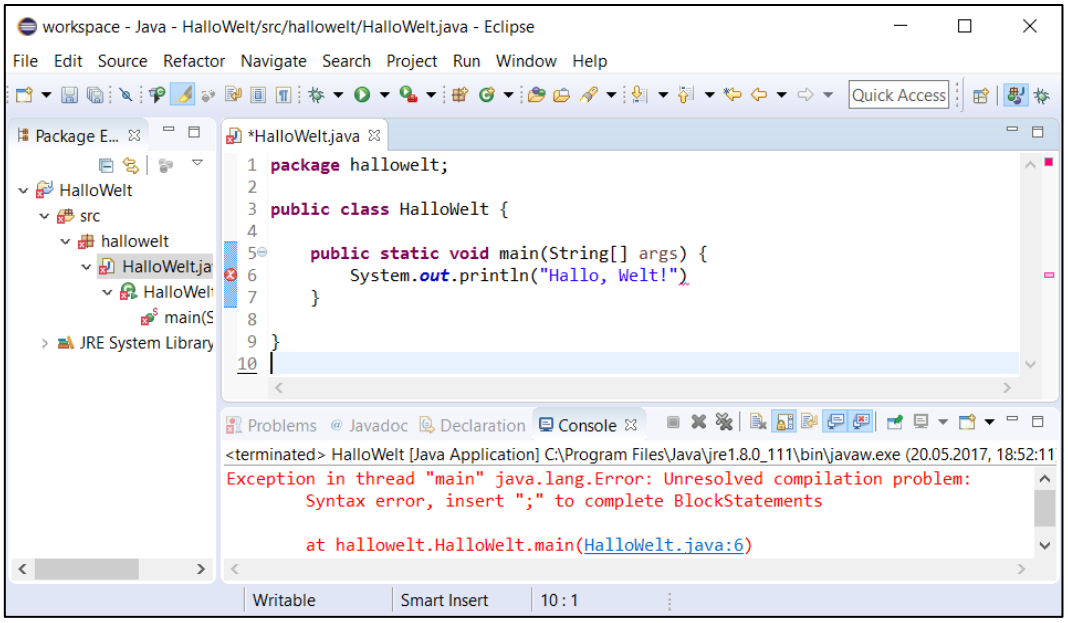


Abb. 6: Die integrierte Entwicklungsumgebung Eclipse

1.3.3 Die Java-IDE der Lemonline AG

Horst Schäfer: „Zu den bekanntesten Java-IDE zählen Eclipse, Netbeans und IntelliJ IDEA. Davon können aber nur Eclipse und Netbeans komplett kostenfrei verwendet werden. Alle drei IDE bieten einen ähnlichen Funktionsumfang. Hier bei der Lemonline AG haben wir uns dafür entschieden, mit Eclipse zu arbeiten. Ihre erste Aufgabe ist es also, Eclipse auf Ihrem Arbeitsrechner zu installieren.“

1.3.4 Installation von Eclipse

Die Installation von Eclipse ist schnell abgeschlossen. Bitte führen Sie die folgenden Schritte durch:

1. Laden Sie Eclipse passend für ihr Betriebssystem herunter (Link: *http://www.eclipse.org/downloads/eclipse-packages/*). Die verschiedenen Eclipse-Versionen unterscheiden sich in ihrer Anzahl an vorinstallierten Plug-ins. Wählen Sie für den Einstieg am besten die Version "Eclipse IDE for Java Developers". Alle Plug-ins, die in dieser Version nicht enthalten sind, können Sie bei Bedarf problemlos nachinstallieren.

Abb. 7: Logo Eclipse IDE for Java Developers

2. Legen Sie in Ihrem Programme-Ordner unter *C:\Program Files* einen Ordner für Eclipse an: *C:\Program Files\Eclipse*.
3. Entpacken Sie die heruntergeladene Datei in Ihrem Eclipse Ordner (*C:\Program Files\Eclipse*).
4. Im Anschluss daran finden Sie in Ihrem entpackten Ordner die ausführbare Datei „eclipse.exe“, mit der Sie Eclipse starten können.
5. Bei Problemen finden Sie Hilfe in der Installationsanleitung (Link: *https://wiki.eclipse.org/Eclipse/Installation*).

1.3.5 Vorbereitungen zur Verwendung von Eclipse

Bevor Sie mit der Entwicklung von Java-Programmen mit Eclipse starten können, müssen einige Einstellungen vorgenommen werden. Bitte führen Sie die folgenden Schritte durch:

1. Öffnen Sie Eclipse mit einem Doppelklick auf die ausführbare Datei eclipse.exe in Ihrem entpackten Eclipse-Ordner.

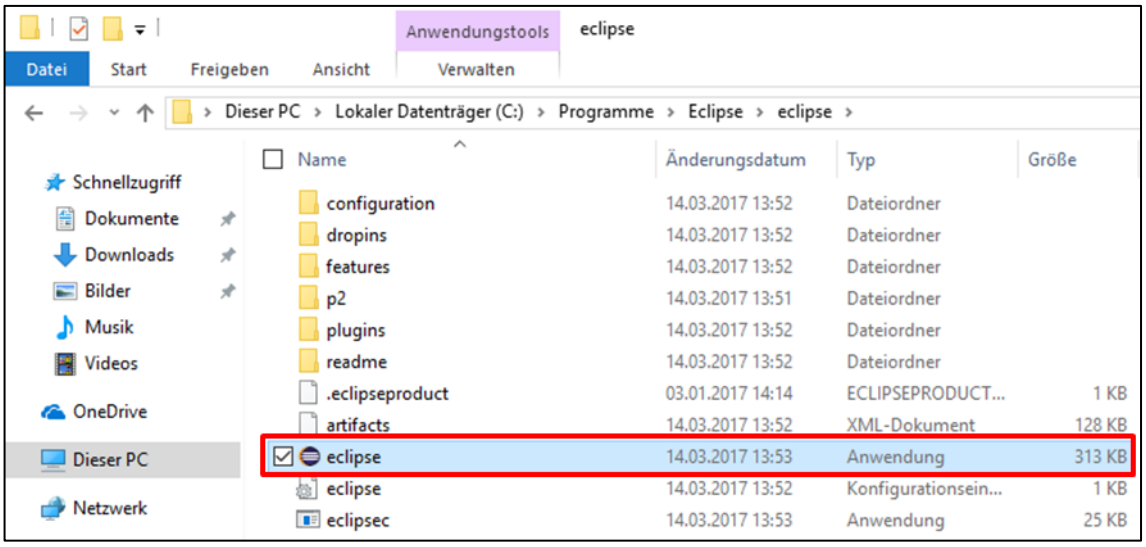


Abb. 8: Öffnen der Datei „eclipse.exe“

2. Es öffnet sich ein Hinweisfenster, in dem Sie den Workspace auf ein Standard-Verzeichnis (z.B. C:\Workspace) festlegen können. Im Workspace werden alle Java-Projekte, die Sie im Laufe dieser WBT-Serie anlegen, abgespeichert.

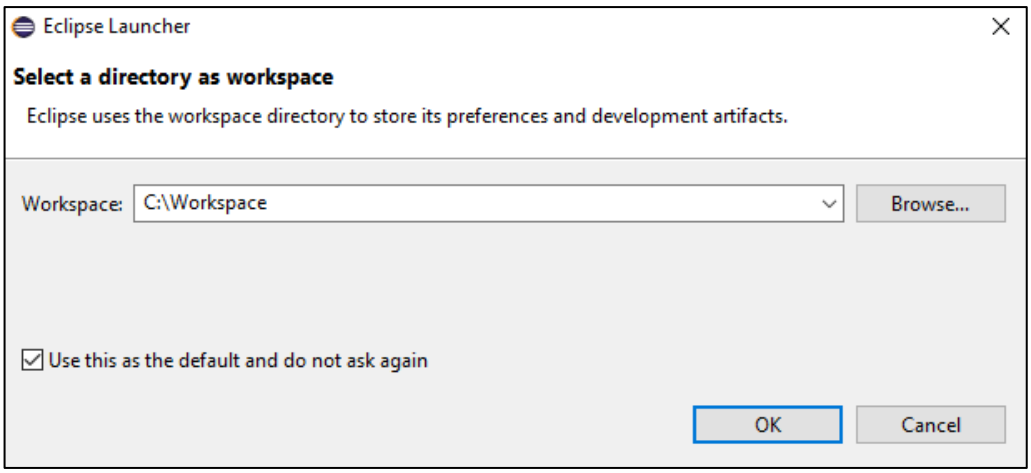


Abb. 9: Standard-Verzeichnis des Workspace festlegen

3. Danach erscheint das Eclipse-Begrüßungsfenster, das Sie für den nächsten Start durch die Schaltfläche in der rechten unteren Ecke deaktivieren können. Durch einen Klick auf „Workbench" in der rechten oberen Ecke gelangen Sie zum Arbeitsbereich von Eclipse.

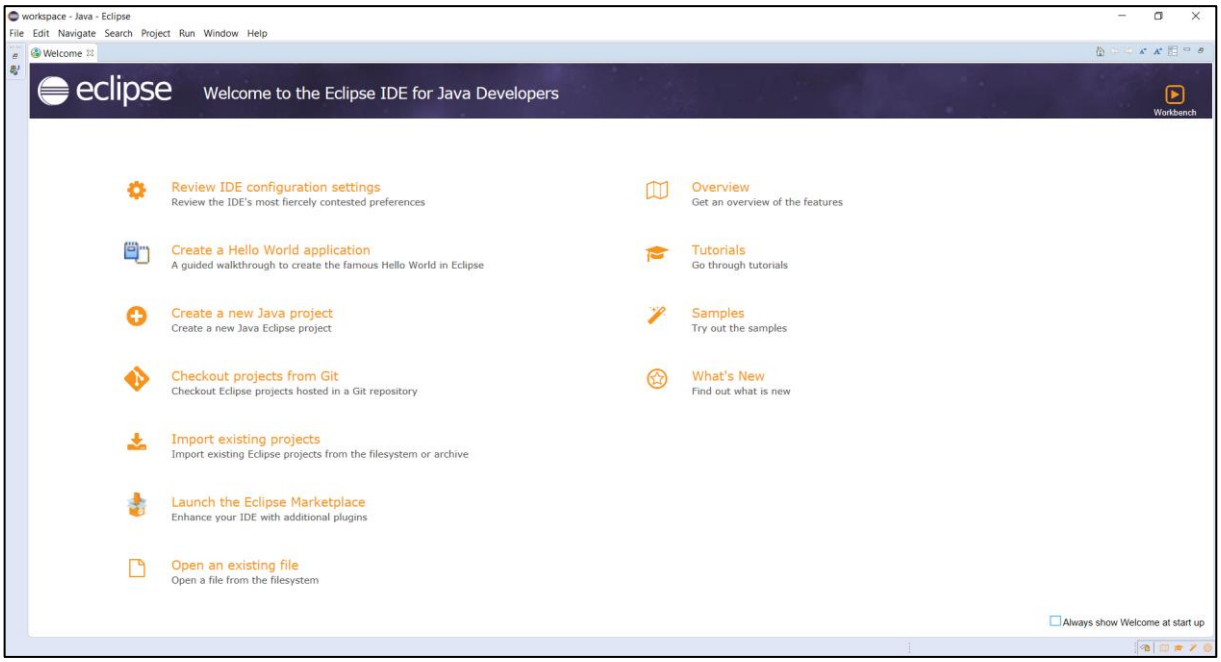


Abb. 10: Eclipse-Begrüßungsfenster

1.3.6 Der Arbeitsbereich

Der Arbeitsbereich von Eclipse heißt Workbench. Die Workbench besteht aus verschiedenen Fenstern, die in Eclipse als Views bezeichnet werden. Die Anordnung der Views nennt man

Perspektive. Die Standard-Perspektive in Eclipse heißt Java-Perspektive. In der Java-Perspektive werden alle Views für die Editierung von Java-Programmen zusammengefasst.

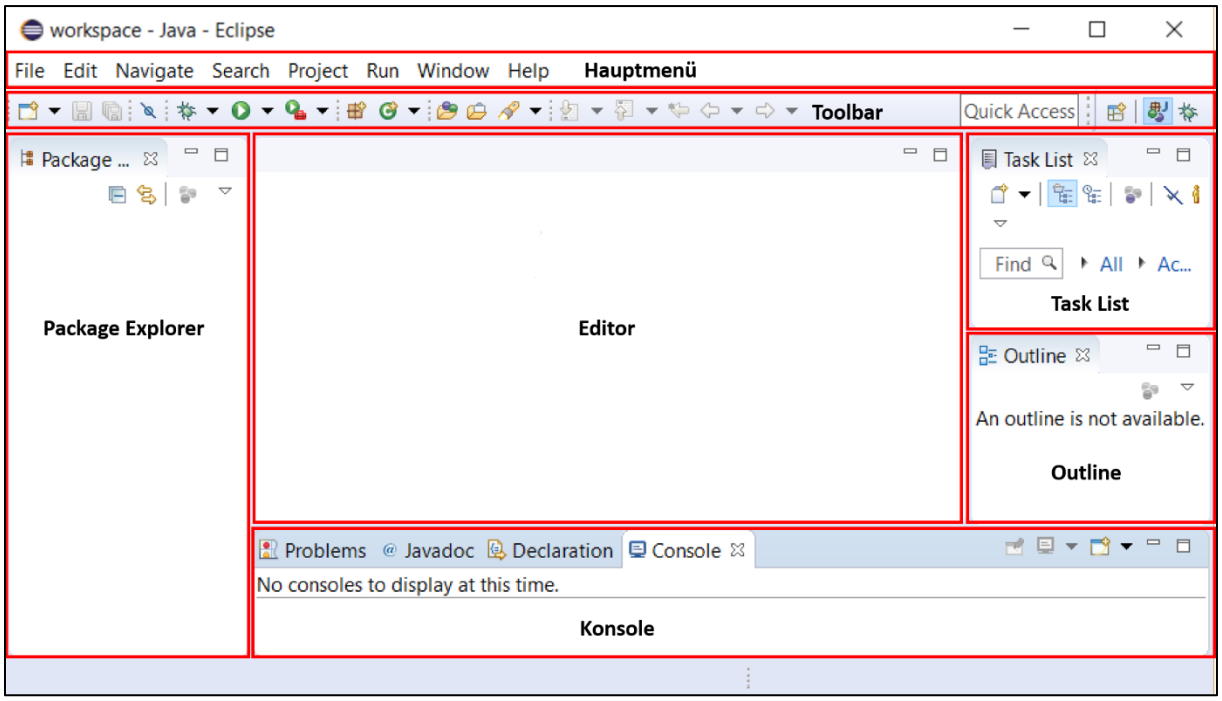


Abb. 11: Die Views der Java-Perspektive

Die Views der Java-Perspektive:

- **Hauptmenü**: Über das Hauptmenü können Sie alle Funktionen von Eclipse erreichen.
- **Toolbar**: Die Toolbar enthält Symbole für den Schnellzugriff auf ausgewählte Funktionen. Diese Funktionen sind auch über das Hauptmenü erreichbar.
- **Package Explorer**: Der Package Explorer dient zur Verwaltung der angelegten Projekte. Alle Pakete und Klasse eines Projektes werden im Package Explorer in einer Hierarchie dargestellt.
- **Editor**: Im Editor wird der Quellcode eines Java-Programms erstellt. Der Quellcode wird automatisch farbig hervorgehoben. Beim Erstellen des Quellcodes werden zukünftige Kompilierfehler rot markiert und ggfs. Lösungsvorschläge zur Behebung des Fehlers angezeigt.
- **Task List**: In der Task List können offene Aufgaben als Tasks angelegt und in einer Übersicht eingesehen werden. Die Task List wird in dieser WBT-Serie nicht genutzt und kann daher über das "x" in der rechten oberen Ecke geschlossen werden.
- **Outline**: Die Outline bietet eine Übersicht über die im Editor geöffnete Java-Datei. In der Outline werden z. B. die Elemente einer Klasse wie die Variablen und Methoden angezeigt. Die Outline-View wird in dieser WBT-Serie nicht genutzt und kann daher über das "x" in der rechten oberen Ecke geschlossen werden.

- **Konsole**: Wenn ein Java-Programm mit Eclipse ausgeführt wird, erscheint dessen Ausgabe in der Konsole. Falls es beim Kompilieren des Java-Programms zu einem Fehler kommt, wird dieser auch in diesem Bereich angezeigt.

1.3.7 Das "HalloWelt"-Programm mit Eclipse

Horst Schäfer: „Nachdem Sie sich an diesem Morgen mit der Workbench vertraut gemacht haben, werde ich Ihnen am Nachmittag zeigen, wie in Eclipse Java-Programme erstellt und ausgeführt werden. Sie kennen aus Ihrem Java-Grundlagenkurs sicherlich schon das "HalloWelt"-Programm. Dieses wollen wir gemeinsam in Eclipse erstellen. Zur Vorbereitung darauf, sollten Sie sich einen Überblick über den Funktionsumfang von Eclipse verschaffen.“

1.4 Java-Programmierung mit Eclipse

1.4.1 Funktionsumfang von Eclipse

Ganz oben in der Workbench befindet sich das Hauptmenü, über das Sie alle Funktionen in Eclipse erreichen können. Eine IDE hat einen sehr großen Funktionsumfang. Einen Teil dieser Funktionen werden Sie im Laufe Ihres Praktikums kennenlernen. Das Hauptmenü umfasst die Registerkarten File, Edit, Source, Refactor, Navigate, Search, Project, Run und Window.

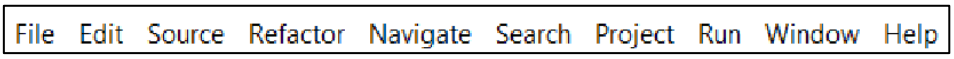


Abb. 12: Das Hauptmenü von Eclipse

1.4.2 Das erste Java-Programm mit Eclipse

Mithilfe des Hauptmenüs werden wir jetzt das "HalloWelt"-Programm mit Eclipse erstellen. Dafür wird zunächst ein sogenanntes Java-Projekt angelegt, das Java-Programm und seine Ressourcen beinhaltet. Danach wird ein Paket und eine Klasse für das "HalloWelt"-Programm erstellt.

Bitte führen Sie die folgenden Schritte durch:

1. Zuerst erzeugen Sie über File/New/New Java Project ein neues Java-Projekt mit dem Namen "HalloWelt".
2. Erstellen Sie danach über File/New/Package im Java-Projekt "HalloWelt" das Paket "hallowelt".
3. Erzeugen Sie in diesem Paket über File/New/Class eine Klasse mit dem Namen "HalloWelt".

Mit dem Java-Projekt, dem Paket und der Klasse haben Sie das Grundgerüst für das "HalloWelt"-Programm angelegt. Jetzt können Sie den Quellcode eingeben und das "HalloWelt"-Programm ausführen.

Bitte führen Sie die folgenden beiden Schritte durch:

1. Fügen Sie in die Klasse *HalloWelt* im Editor den Quellcode ein:

```
public static void main(String[] args) {
  System.out.println("Hallo, Welt!");
}
```

2. Jetzt können Sie das "HalloWelt"-Programm über Run/Run ausführen. Es erscheint der Satz "Hallo, Welt!" in der Konsole.

Hinweis: Die Abbildung 13 entspricht im WBT einem Video zur Verdeutlichung der Inhalte.

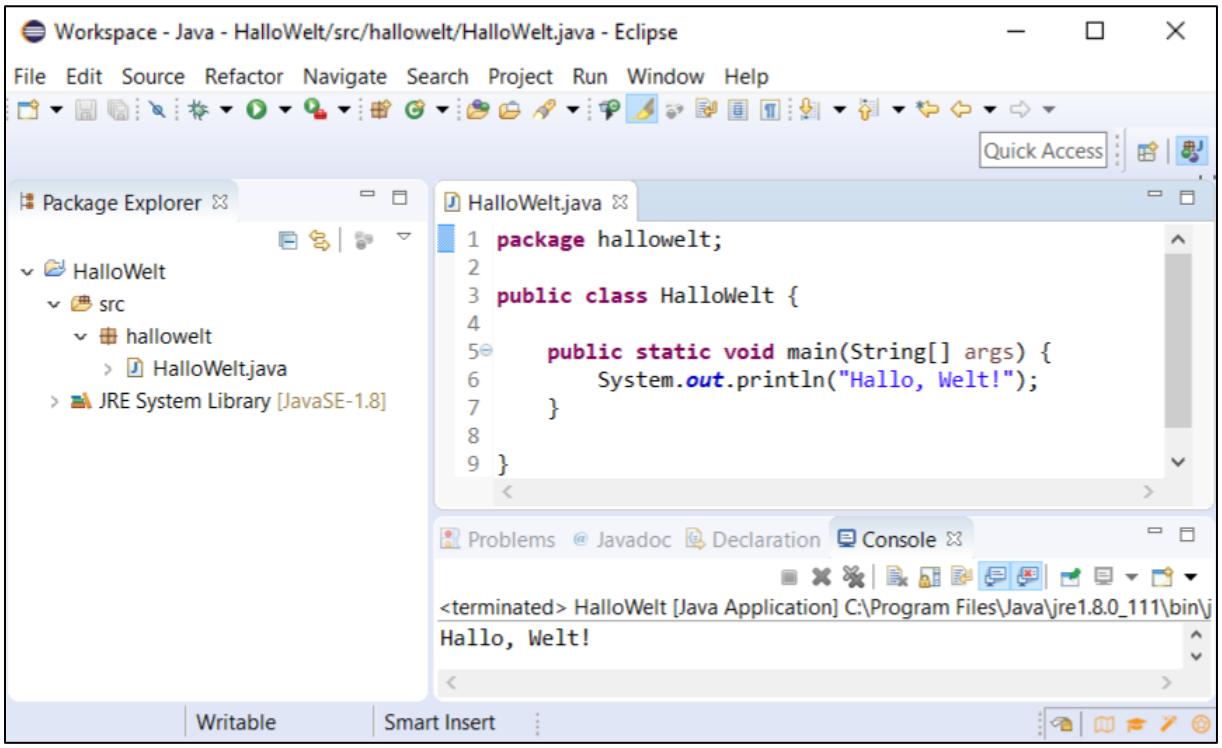


Abb. 13: Das "HalloWelt"-Programm mit Eclipse

1.4.3 Lessons learned

Horst Schäfer: „Herzlichen Glückwunsch! Im Laufe des Tages haben Sie gelernt, was eine IDE ist und wie man Eclipse installiert. Danach haben Sie Ihr erstes Java-Programm mit Eclipse erstellt und ausgeführt. Das ist eine gute Grundlage, für Ihre kommenden Aufgaben. Als nächstes werden Sie eine Schulung zum Thema "objektorientierte Programmierung mit Java und Eclipse" erhalten. Wenn Sie weitere Fragen haben, können Sie mich jederzeit kontaktieren.“

2. Erste Objekte mit Eclipse erzeugen

2.1 Der erste Schulungstag

Horst Schäfer: „Hallo Frau Schmitt, ich heiße Sie als Praktikantin in unserer Schulung zum Thema "Objektorientierte Programmierung mit Java und Eclipse" herzlich willkommen! In der Schulung erhalten Sie eine Einführung in die objektorientierte Java-Programmierung mit der IDE Eclipse. Dabei werden Sie die relevanten Grundlagen erlernen und darauf aufbauend objektorientierte Java-Programme mit Eclipse erstellen.“

2.2 Objekte und Objektorientierung

Horst Schäfer: „Bisher haben Sie vor allem Prozeduren programmiert. Das bedeutet, Sie haben Ihre Programme als eine Abfolge von (statischen) Methoden entwickelt und diese aus der main-Methode aufgerufen:

```
public class MatheProgramm  {
    public static int quadriere(int x) {
         return x * x;
    }
    public static void main(String[] args) {
          int zahl = quadriere(5);
          System.out.println(zahl);
    }
}
```

In der objektorientierten Programmierung hingegen werden Programme mit Objekten realisiert. Objekte stehen für Gegenstände, wie z. B. Autos, Menschen und Adressen. Jedes Objekt verfügt über eigene Eigenschaften und Verhaltensweisen. Objekte werden mit ihren Eigenschaften und ihrem Verhalten im Quellcode einer Programmiersprache abgebildet. Dabei werden verschiedene Konzepte berücksichtigt. Die einzelnen Konzepte werden Sie im Laufe dieser Schulung kennenlernen.“

2.3 Das Objekt “Mensch“ abbilden

Um ein Objekt abzubilden, müssen wir uns zunächst überlegen, welche Eigenschaften und Verhaltensweisen es haben soll. Ein Mensch hat beispielsweise einen Vor- und Nachnamen und kann sprechen. Um in Java das Objekt "Mensch" zu realisieren, erstellt man zuerst eine

Klasse mit dem Namen *Mensch*. Diese Klasse dient als Bauplan. Im Bauplan werden Eigenschaften durch Variablen und das Verhalten durch nicht-statische Methoden beschrieben.

Bitte führen Sie die folgenden Schritte durch, um die Klasse *Mensch* mit Eclipse zu erstellen:

1. Bitte erstellen Sie ein neues Java-Projekt mit dem Namen "ObjektorientierteProgrammierung" und ein neues Paket "wbt02". Erzeugen Sie im Paket "wbt02" die Klasse *Mensch*.
2. Fügen Sie in den Klassenkörper der Klasse *Mensch* die Eigenschaften `String vorname` und `String nachname` ein.
3. Schreiben Sie die Methode `sprechen` ohne Rückgabewert, die den folgenden Satz in der Konsole ausgibt: "Hallo, Welt! Ich heiße <vorname> <nachname>.":

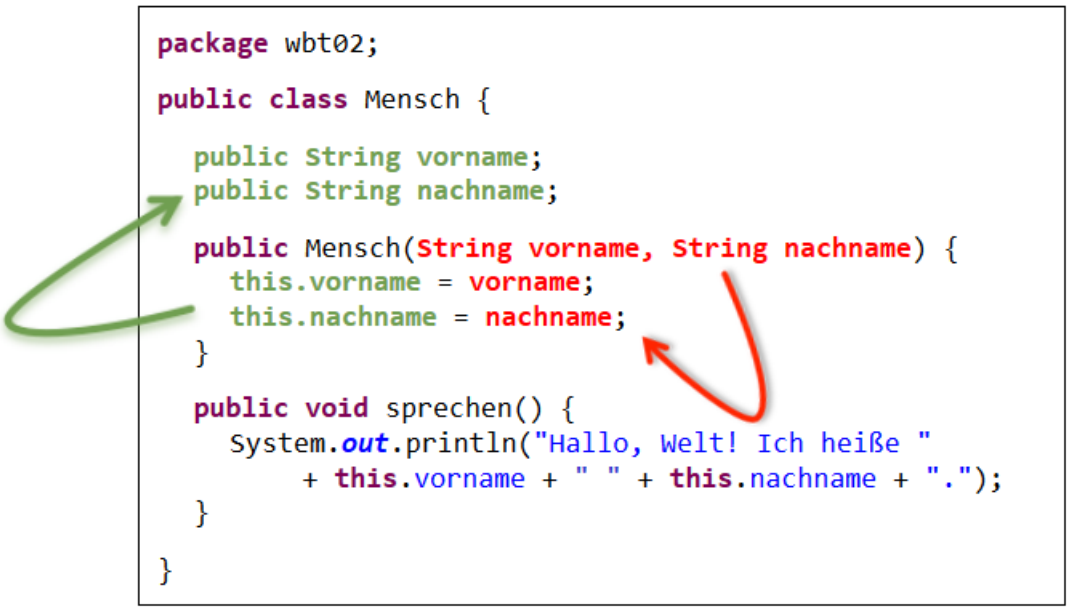
```
package wbt02;

public class Mensch {

	public String vorname;
	public String nachname;

	public Mensch(String vorname, String nachname) {
		this.vorname = vorname;
		this.nachname = nachname;
	}

	public void sprechen() {
		System.out.println("Hallo, Welt! Ich heiße "
			+ this.vorname + " " + this.nachname + ".");
	}

}
```

Was das Schlüsselwort this bedeutet, werden Sie auf der nächsten Seite lernen.

2.4 Der Konstruktor

Damit Menschen ein realer Vor- und Nachname (z. B. Anna Schmitt) zugewiesen werden kann, benötigen wir einen Konstruktor. Ein Konstruktor hat immer den gleichen Namen, wie die Klasse, in der er steht, und runde Klammern für seine Parameter. Parameter sind die Werte, die beim Erzeugen eines Objekts übergeben werden müssen (hier: `String vorname` und `String nachname`).

Um Namenskonflikte zwischen den Parametern des Konstruktors und den Eigenschaften der Klasse zu vermeiden, wird das Schlüsselwort this verwendet. Also z. B.: `this.vorname = vorname;`. Das Schlüsselwort this bezieht sich immer auf die Eigenschaften des Objekts, das die Methode aufgerufen hat.



```
package wbt02;

public class Mensch {

   public String vorname;
   public String nachname;

   public Mensch(String vorname, String nachname) {
      this.vorname = vorname;
      this.nachname = nachname;
   }

   public void sprechen() {
      System.out.println("Hallo, Welt! Ich heiße "
            + this.vorname + " " + this.nachname + ".");
   }

}
```

Abb. 14: Zuweisung der Parameter zu den Eigenschaften im Konstruktor

2.5 Objekte der Klasse Mensch erzeugen

Wir benötigen eine zweite Klasse mit einer main-Methode, um Objekte der Klasse *Mensch* zu erzeugen. Da Menschen auf der Erde leben, erstellen wir in unserem Paket "wbt02" die Klasse *Erde* mit einer main-Methode. Wenn Sie beim Erstellen der Klasse *Erde* die Schaltfläche "public static void main(String[] args)" aktivieren, wird die main-Methode automatisch erzeugt.

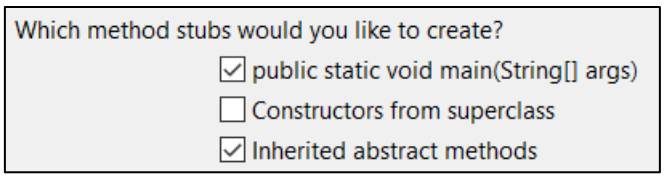


Abb. 15: main-Methode einer Klasse automatisch erzeugen

Als nächstes müssen Variablen definiert werden, in denen wir unsere Objekte speichern. Wir wollen Objekte für die realen Menschen Anna Schmitt und Horst Schäfer erzeugen. Daher nennen wir die Variablen "aSchmitt" und "hSchaefer". Damit wir in den Variablen "aSchmitt" und "hschaefer" ein Objekt der Klasse "Mensch" speichern können, müssen wir beim Erzeugen der Variablen den Datentyp `Mensch` angeben:

```
package wbt02;

public class Erde {

	public static void main(String[] args) {
		Mensch aSchmitt;
		Mensch hSchaefer);
	}

}
```

Typische Datentypen in Java sind `int`, `double` oder `String`. Es können jedoch auch Klassen, die als Baupläne für Objekte geschrieben wurden, als Datentyp verwendet werden. Hier z. B. die Klasse *Mensch*.

Mit dem Schlüsselwort "new" erzeugen wir ein neues Objekt und speichern es in der zuvor definierten Variable:

```
package wbt02;

public class Erde {

	public static void main(String[] args) {
		Mensch aSchmitt = new Mensch("Anna", "Schmitt", 1996);
		Mensch hSchaefer = new Mensch("Horst", "Schaefer", 1970);
	}

}
```

Der Befehl `Mensch aSchmitt = new Mensch("Anna", "Schmitt");` erzeugt z. B. ein Objekt mit dem Vornamen "Anna" und dem Nachnamen "Schmitt". Dieses Objekt wird in der Variable "aSchmitt" gespeichert. Das Schlüsselwort "new" ruft dabei immer den Konstruktor der Klasse auf, die wir als Datentyp verwenden. Hier also den Konstruktor der Klasse *Mensch*.

2.6 Anpassung des Konstruktors

Die Klasse *Mensch* soll nun um die Eigenschaft `int geburtsjahr` erweitert werden, um jedem Menschen ein Geburtsjahr zuzuweisen. Dafür müssen wir auch den Konstruktor der Klasse *Mensch* anpassen und das Geburtsjahr als Parameter hinzufügen:

```
package wbt02;

public class Mensch {

	public String vorname;
	public String nachname;
	public int geburtsjahr;

	public Mensch(String vorname, String nachname, int geburtsjahr){
		this.vorname = vorname;
		this.nachname = nachname;
		this.geburtsjahr = geburtsjahr;
	}

	public void sprechen() {
		System.out.println("Hallo, Welt! Ich heiße "
			+ this.vorname + " " + this.nachname + ".");
	}

}
```

Wenn wir jetzt Objekte der Klasse Mensch erzeugen, müssen wir im new-Befehl nicht mehr nur den Vor- und Nachnamen, sondern auch das Geburtsjahr als Parameter übergeben. Anna Schmitt wurde im Jahr 1996 und Horst Schäfer 1970 geboren:

```
package wbt02;

public class Erde {

	public static void main(String[] args) {
		Mensch aSchmitt = new Mensch("Anna", "Schmitt", 1996);
		Mensch hSchaefer = new Mensch("Horst", "Schaefer", 1970);
	}

}
```

2.7 Klassen als Bauplan für Objekte

Bisher haben wir den Datentyp `Mensch` in der Klasse *Mensch* definiert und in der Klasse *Erde* die beiden Variablen "aSchmitt" und "hSchaefer" vom Datentyp `Mensch` erzeugt. Die Objekte, die in den Variablen "aSchmitt" und "hSchaefer" gespeichert wurden, bestehen aus einer exakten Kopie der Eigenschaften und Verhaltensweisen, die in der Klasse *Mensch* definiert wurden. Wir verwenden die Klasse *Mensch* also als Bauplan. Dabei haben wir z. B. mit `Mensch aSchmitt` die Variable "aSchmitt" vom Datentyp `Mensch` angelegt. Mit `new Mensch ("Anna", "Schmitt", 1996)` wurde ein neues Objekt erzeugt und über den Aufruf des Konstruktors der Vorname "Anna", der Nachname "Schmitt" und das Geburtsjahr "1996" zugewiesen.

2.8 Objektmethoden aufrufen

In die Klasse *Mensch* haben wir die nicht-statische Methode (eine sog. Objektmethode) `public void sprechen()` geschrieben. Unsere Objekte, die in den Variablen "aSchmitt" und "hSchaefer" gespeichert wurden, bestehen aus einer exakten Kopie aller Eigenschaften und Verhaltensweisen der Klasse *Mensch*. Daher können sie sich über ihre sprechen()-Methode mit Ihrem realen Vorund Nachnamen vorstellen. Objektmethoden werden mit der Punktnotation über die Variablen, in denen das Objekt gespeichert ist, aufgerufen (hier "aSchmitt" und "hSchaefer"). Die Befehle lauten:

- `aSchmitt.sprechen();`
- `hSchaefer.sprechen();`

```
package wbt02;

public class Erde {

 public static void main(String[] args) {
       Mensch aSchmitt = new Mensch("Anna", "Schmitt", 1996);
       Mensch hSchaefer = new Mensch("Horst", "Schaefer", 1970);

       aSchmitt.sprechen();
       hSchaefer.sprechen();
 }

}
```

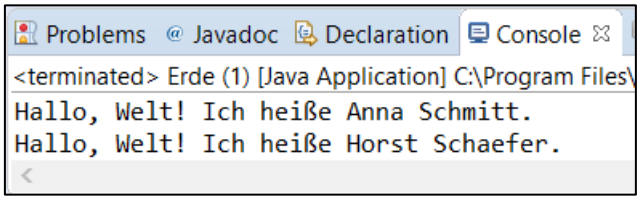


Abb. 16: Objektmethoden der Klasse „Mensch“ ausführen

2.9 Objektorientierte Modellierung

Nachdem Sie Ihr erstes Objekt mit Eclipse erzeugt und verwendet haben, schauen wir uns jetzt die Standardmodellierungssprache für objektorientierte Programme an, die Unified Modeling Language (kurz UML). Die UML wurde von den drei sogenannten "Amigos" Grady Booch, Ivar Jacobson und James Rumbough entwickelt:

- **1996**: Im Jahr 1996 führten die drei "Amigos" ihre Methoden zur "Unified Modeling Language" zusammen. Diese wurde aufgrund ihrer Popularität zu einem Quasi-Standard.
- **1997**: In der Version 1.1 wurde die UML im Jahr 1997 bei der Object Management Group (OMG) eingereicht und als Standard akzeptiert.
- **2005**: Im Jahr 2005 wurde die UML 2.0 mit wesentlichen Veränderungen eingeführt. Zuvor wurde die UML kontinuierlich in den Versionen 1.2 - 1.5 weiterentwickelt.
- **2015**: Im Jahr 2005 wurde der bis heute gültige Standard UML 2.5 eingeführt.

Die UML besteht aus UML-eigenen Notationselementen, mit denen verschiedene Diagramme erzeugt werden können. Es gibt keine abgeschlossene Diagrammübersicht, da die Notationselemente beliebig zu einem Diagramm zusammengesetzt werden können. Zudem existiert kein vorgegebener Prozess, welches Diagramm zu welchem Zeitpunkt erzeugt werden sollte.

2.10 UML-Klassendiagramm

Wir arbeiten hier vor allem mit sogenannten UML-Klassendiagrammen. Mit UML-Klassendigrammen wird die statische Struktur einer Java-Klasse und deren Beziehung zu anderen Java-Klassen visualisiert. In einem UML-Klassendiagramm wird eine Klasse als Rechteckt dargestellt, das den Namen der Klasse enthält. Zusätzlich kann das Rechteck auch in drei Bereiche unterteilt werden. Das Rechteck beinhaltet unter dem Namen, auch die Eigenschaften und Methoden der Klasse.

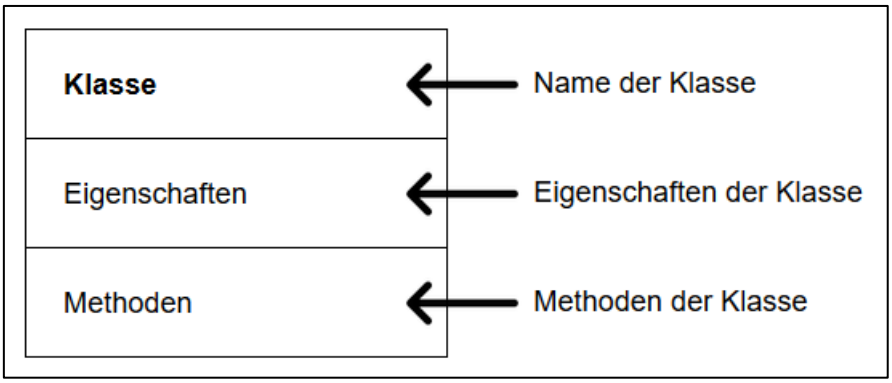


Abb. 17: Bestandteile eines UML-Klassendiagramms

Ab dem zweiten Schulungstag erhalten Sie UML-Klassendiagramme und werden daraus Java-Programme entwickeln.

2.11 Vom UML-Klassendiagramm zum Java-Quellcode

Auf Basis von einem UML-Klassendiagramm kann der Quellcode für die Java-Klasse *Mensch* geschrieben werden. Dafür müssen die einzelnen Bereiche des UML-Klassendiagramms in die Klasse übertragen werden. Eigenschaften und Methoden, denen im UML-Klassendiagramm ein Pluszeichen (+) vorangestellt wird, werden in der Java-Klasse als `public` deklariert. Ein Minuszeichen (-) bedeutet, dass die Eigenschaften und Methoden als `private` deklariert werden.

- Der erste Bereich des UML-Klassendiagramms beinhaltet den Namen der Klasse.
- Im zweiten Bereich befinden sich die Eigenschaften. Diese werden zuerst in der Klasse mit ihren Datentypen deklariert.
- Der untere Bereich des UML-Klassendiagramms beinhaltet den Konstruktor und die Methoden der Klasse. Der Konstruktor und die Methoden werden unterhalb von den Eigenschaften in die Klasse Mensch geschrieben.

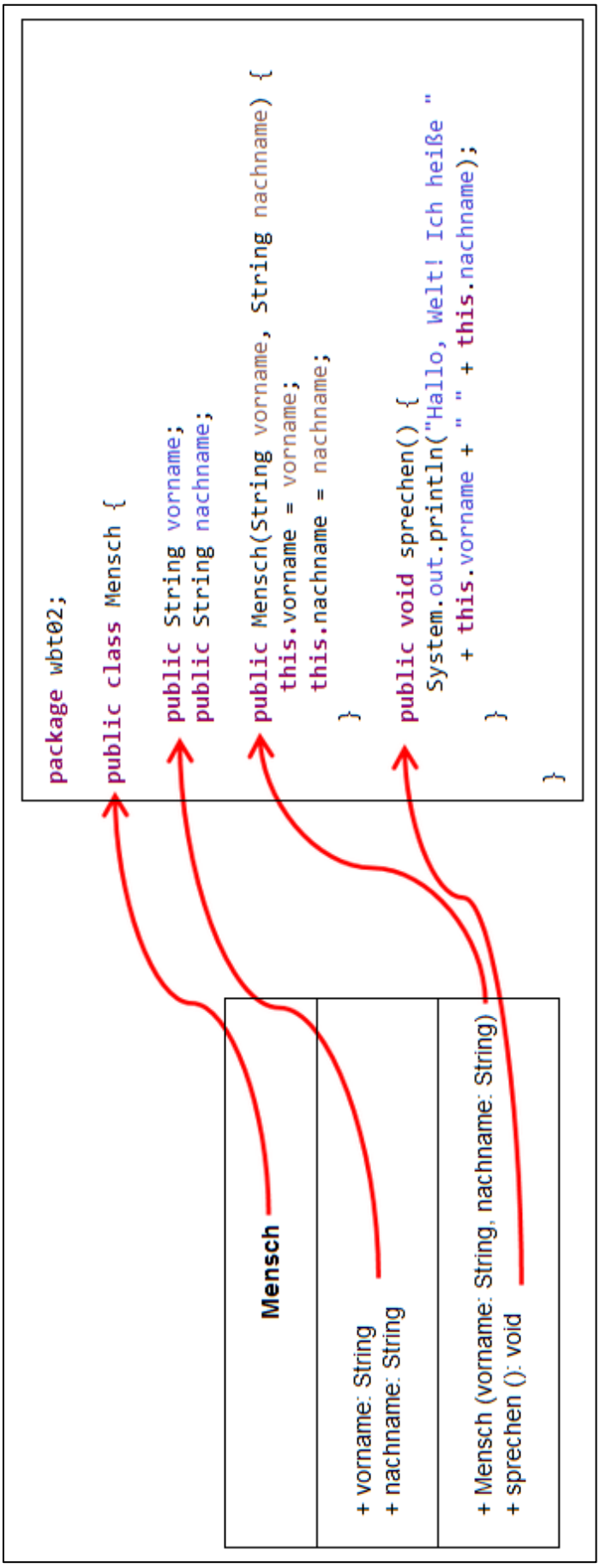


Abb. 18: Vom UML-Klassendiagramm zum Java-Quellcode

2.12 UML-Anwendungsfalldiagramm

Klassendiagramme stellen die statische Struktur eines Programms dar und zählen damit zu den sog. Strukturdiagrammen. Die zweite Kategorie für UML Diagramme sind die sog. Verhaltensdiagramme. In Verhaltensdiagrammen werden die dynamischen Aspekte eines Programms dargestellt. Zu den Verhaltensdiagrammen zählt z. B. das UML-Anwendungsfalldiagramm. UML-Anwendungsfalldiagramme geben eine Gesamtübersicht über ein Programm mit seiner Funktionalität und den beteiligten Akteuren. Dafür werden die konkreten Anwendungsfälle mit ihrer Beziehung zu den beteiligten Akteuren innerhalb eines Systems dargestellt.

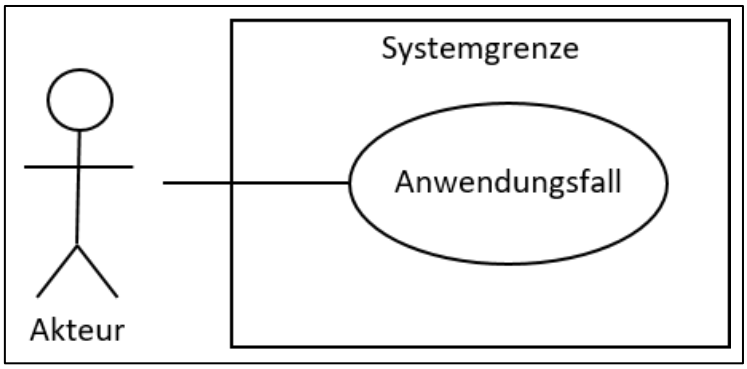


Abb. 19: Bestandteile eines UML-Anwendungsfalldiagramms

Beispielsweise kann ein Kunde (= Akteur) von einem Geldautomaten (= Systemgrenze) Geld abheben (= Anwendungsfall).

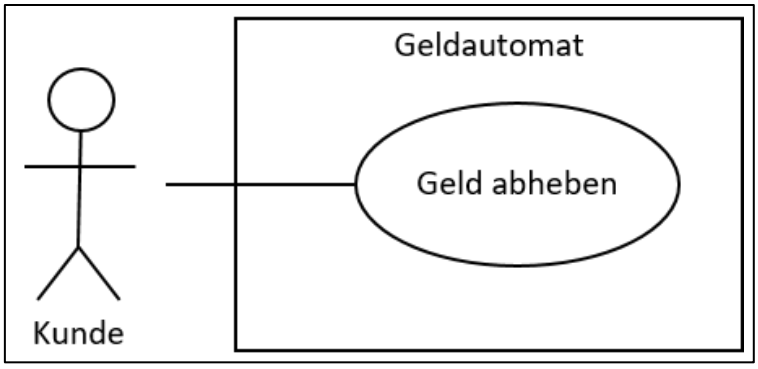


Abb. 20: Beispiel eines UML-Anwendungsfalldiagramms

2.13 Lessons learned

Am ersten Schulungstag haben Sie gelernt, wie Objekte beschrieben und in Java realisiert werden. Dabei haben Sie die Standardmodellierungssprache für objektorientierte Programme UML kennengelernt. Sie haben gelernt, dass Sie Ihre Objekte zuerst in Klassen planen müssen. Zur Visualisierung nutzen wir sogenannte UML-Klassendiagramme. Auf Basis dieser UML-Klassendiagramme wird für die entsprechende Klasse der Java-Quellcode erstellt. Dabei dient die Klasse als Bauplan für ein Objekt. Um ein Objekt aus einer Klasse zu erzeugen, braucht man eine zweite Klasse mit einer main-Methode.

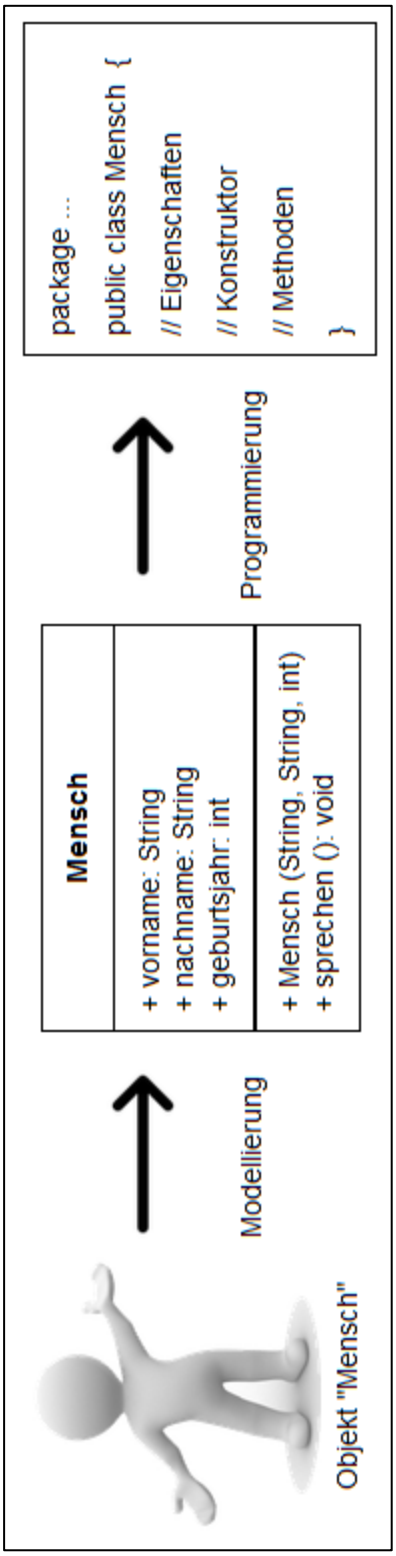


Abb. 21: Von der Planung eines Objekts zum Java-Quellcode

3. Vererbung am Beispiel von Studierenden und Professoren

3.1 Der zweite Schulungstag

Horst Schäfer: „Hallo Frau Schmitt, ich heiße Sie als Praktikantin zum zweiten Tag unserer Schulung "Objektorientierte Programmierung mit Java und Eclipse" herzlich willkommen! Am ersten Schulungstag haben wir bereits Objekte aus der Klasse *Mensch* erzeugt. Heute wollen wir auf Basis der Klasse *Mensch* die Klassen *Studierender* und *Professor* schreiben. Dabei nutzen wir das Konzept der Vererbung.“

3.2 Vereinfachtes Beispiel

Bevor wir die Klassen *Studierender* und *Professor* schreiben, müssen wir diese planen. Wir überlegen uns also, welche Eigenschaften und Verhaltensweisen Studierende und Professoren haben. Studierende haben z. B. einen Vor- und Nachnamen, ein Geburtsjahr und eine Matrikelnummer. Zudem können Studierende sprechen und Klausuren schreiben. Professoren haben z. B. einen Vor- und Nachnamen, ein Geburtsjahr und eine Personalnummer. Außerdem können Professoren sprechen und Klausuren korrigieren.

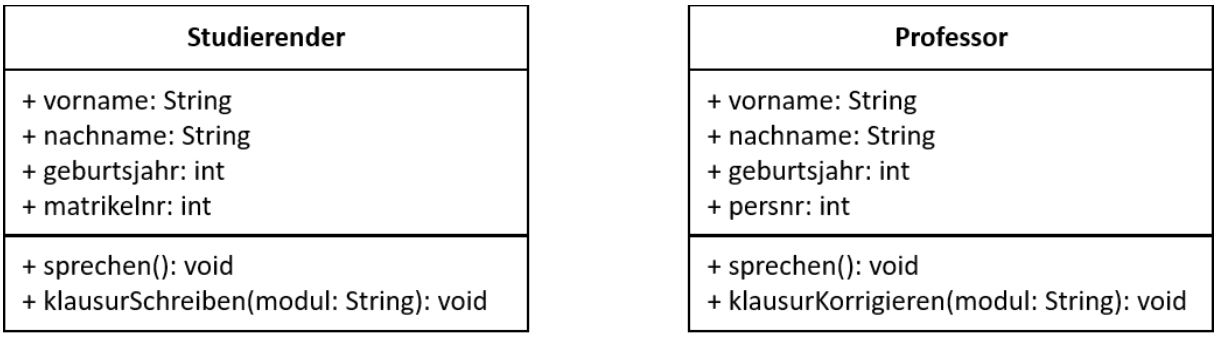


Abb. 22: UML-Klassendiagramme der Klassen „Studierender“ und „Professor“

Wenn Sie jetzt die Baupläne von Studierenden und Professoren miteinander vergleichen, werden Sie feststellen, dass die Eigenschaften und Verhaltensweisen teilweise übereinstimmen.

3.3 Vererbung

Um gleiche Eigenschaften und Verhaltensweisen nicht doppelt im Quellcode schreiben zu müssen, fassen wir diese in einer neuen Klasse zusammen. Wir verwenden dafür die Klasse *Mensch*. Damit Studierende und Professoren die Eigenschaften (Name, Vorname, Geburtsjahr) und Verhaltensweisen (sprechen) der Menschen nutzen können, müssen sie diese erben. Wie das in Java funktioniert, werden Sie auf den nächsten Seiten lernen.

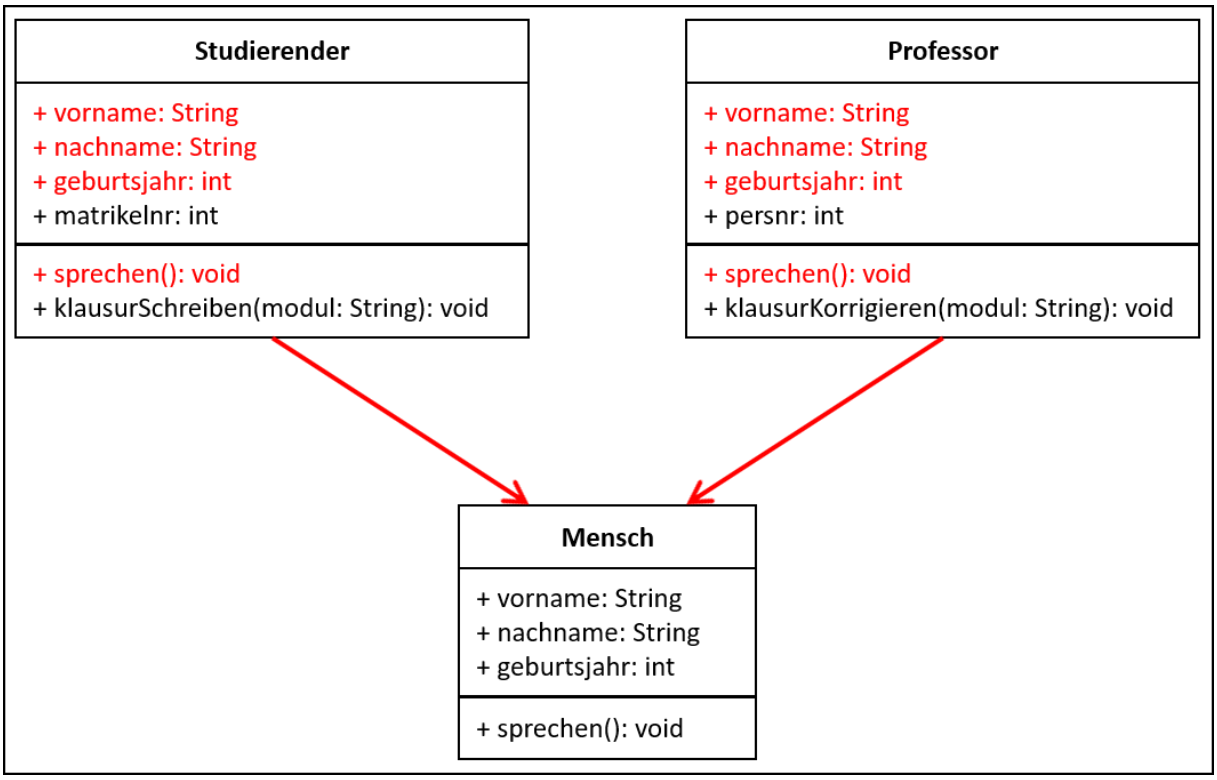


Abb. 23: Vererbung am Beispiel von Menschen, Studierenden und Professoren

3.4 Vererbung im UML-Klassendiagramm

Im UML-Klassendiagramm wird die Vererbung durch einen Pfeil dargestellt. Der Pfeil beginnt an der sog. Subklasse und zeigt mit der Pfeilspitze auf die sog. Superklasse:

- **Superklasse**: Eine Superklasse vererbt ihre Eigenschaften und Verhaltensweise an ihre Subklassen.
- **Subklasse**: Eine Subklasse erbt alle Eigenschaften und Verhaltensweisen von ihrer Superklasse. Zusätzlich kann eine Subklasse weitere Eigenschaften und Verhaltensweisen haben.

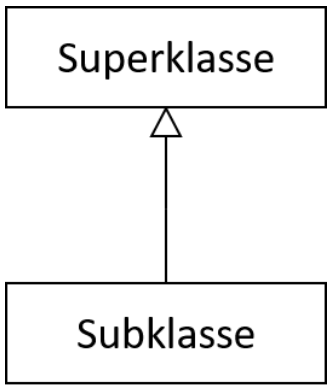


Abb. 24: Vererbung im UML-Klassendiagramm

In unserem Beispiel ist die Klasse *Mensch* unsere Superklasse, von der die Subklassen *Studierender* und *Professor* alle Eigenschaften und Verhaltensweisen erben.

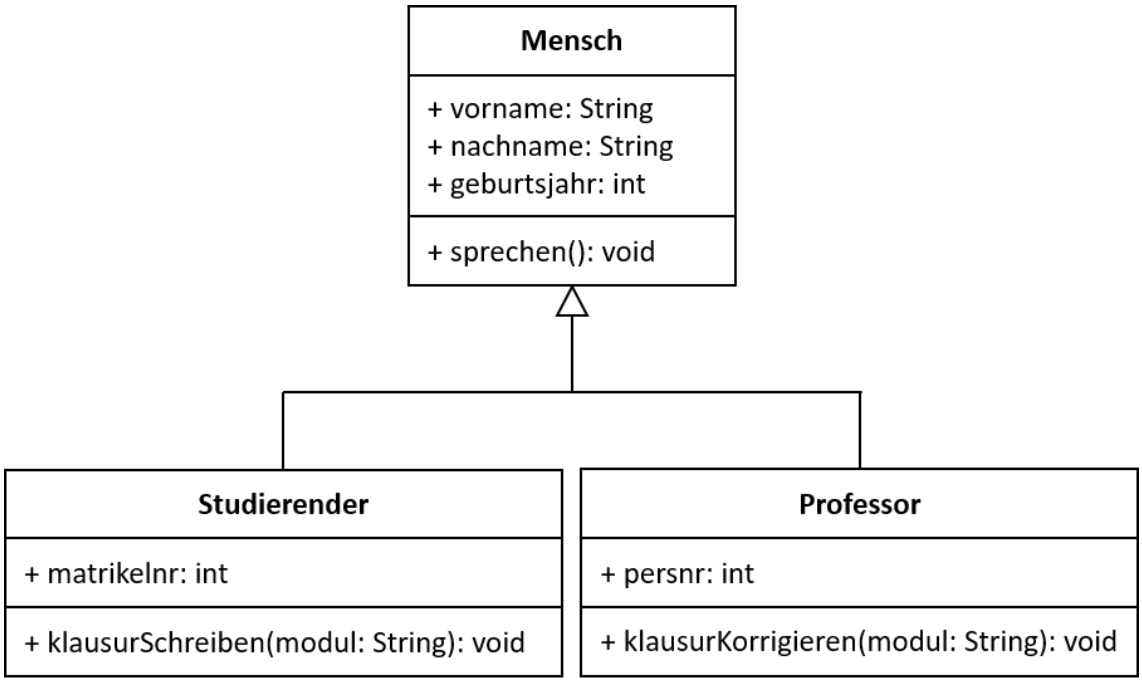


Abb. 25: Vererbung der Klasse „Mensch“ im UML-Klassendiagramm

3.5 Vererbung in Java

In unserem Java-Projekt "ObjektorientierteProgrammierung" erzeugen wir zuerst das Paket "wbt03". Im Paket "wbt03" erstellen wir die Klassen *Mensch*, *Studierender* und *Professor*.

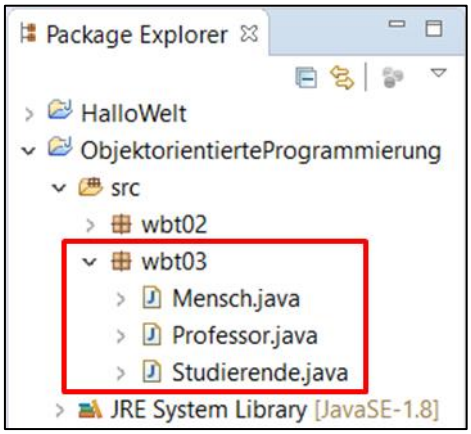


Abb. 26: Das Paket "wbt03"

Damit die Klassen *Studierender* und *Professor* von der Klasse *Mensch* erben können, verwenden wir das Schlüsselwort extends:

```
package wbt03;

public class Studierender extends Mensch {

      public int matrikelnr;

}
```

```
package wbt03;

public class Professor extends Mensch {

	public int persnr;

}
```

Durch das Schlüsselwort extends erben die Klassen *Studierender* und *Professor* alle Eigenschaften und Verhaltensweisen der Klasse *Mensch*. Den Quellcode der Klasse Mensch kennen Sie bereits vom ersten Schulungstag:

```
package wbt03;

public class Mensch {

	public String vorname;
	public String nachname;
	public int geburtsjahr;

	public Mensch(String vorname, String nachname, int geburtsjahr){
		this.vorname = vorname;
		this.nachname = nachname;
		this.geburtsjahr = geburtsjahr;
	}

	public void sprechen() {
		System.out.println("Hallo, Welt! Ich heiße "
			+ this.vorname + " " + this.nachname + ".");
	}

}
```

3.6 Konstruktoren in der Vererbung

Damit Studierenden und Professoren Eigenschaften zugewiesen werden können, benötigen wir einen Konstruktor. Wie Sie am ersten Schulungstag gelernt haben, hat der Konstruktor immer den gleichen Namen wie die Klasse, in der er steht, und runde Klammern für seine Parameter. Im Block des Konstruktors der Klassen *Studierender* und *Professor* werden den Eigenschaften der Klasse die Parameter des Konstruktors zugewiesen.

Im Gegensatz zu Eigenschaften und Verhaltensweisen wird der Konstruktor der Superklasse *Mensch* nicht vererbt. Um Studierenden und Professoren trotzdem einen realen Vor- und Nachnamen und ein Geburtsjahr zuzuweisen, müssen wir direkt auf den Konstruktor der Klasse *Mensch* zugreifen. Dafür verwenden wir den super()-Befehl und schreiben die Parameter des Konstruktors der Superklasse in die runden Klammern. Zudem müssen wir den Vor- und Nachnamen sowie das Geburtsjahr zu den Parametern der Konstruktoren für Studierende und Professoren hinzufügen:

```java
package wbt03;

public class Studierender extends Mensch {

	public int matrikelnr;

	public Studierender(String vorname, String nachname,
	    int geburtsjahr, int matrikelnr) {
	      super(vorname, nachname, geburtsjahr);
	      this.matrikelnr = matrikelnr;
	}

}
```

```java
package wbt03;

public class Professor extends Mensch {

	public int persnr;

	public Professor(String vorname, String nachname,
	    int geburtsjahr, int persnr) {
	      super(vorname, nachname, geburtsjahr);
	      this.persnr = persnr;
	}

}
```

3.7 Studierende und Professoren erzeugen

Nachdem wir den Konstruktor für Studierende und Professoren geschrieben haben, können wir zum ersten Mal Objekte aus den Klassen *Studierender* und *Professor* erzeugen. Dafür erstellen wir die Klasse *Universitaet*. Als erstes erzeugen wir den Studierenden Tim Faber mit dem Geburtsjahr 1995 und der Matrikelnummer 1234. Wir speichern dieses Objekt in der Variablen "tFaber". Danach erzeugen wir den Professor Carl Fink mit dem Geburtsjahr 1965 und der Personalnummer 1. Wir speichern dieses Objekt in der Variablen "cFink". Da Studierende und Professoren von der Klasse *Mensch* erben, können Sie mit der Punktnotation direkt nach Ihrer Erzeugung auf die Methode `public void sprechen()` zugreifen:

```java
package wbt03;

public class Universitaet {

	public static void main(String[] args) {
		Studierender tFaber = new Studierender("Tim", "Faber",
		    1995, 1234);
		Professor cFink = new Professor("Carl", "Fink", 1965, 1);
		tFaber.sprechen();
		cFink.sprechen();
	}
}
```

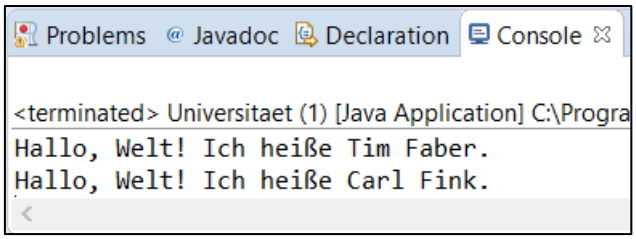


Abb. 27: Aufruf der Objektmethoden der Superklasse durch Objekte der Subklassen

3.8 Objektmethoden der Subklassen

Nachdem wir Objekte aus den Klassen *Studierender* und *Professor* erzeugt haben, wollen wir sie um ihre spezifischen Verhaltensweisen erweitern. Studierende sollen die Möglichkeit erhalten, an einer Klausur teilzunehmen. Dafür schreiben wir die Methode `public void klausurSchreiben(String modul)`:

```
package wbt03;

public class Studierender extends Mensch {

	public int matrikelnr;

	public Studierender(String vorname, String nachname,
	    int geburtsjahr, int matrikelnr) {
	      super(vorname, nachname, geburtsjahr);
	      this.matrikelnr = matrikelnr;
	}

	public void klausurSchreiben(String modul) {
		System.out.println(this.vorname + " " + this.nachname
		     + " schreibt die Klausur " + modul + ".");
	}

}
```

Professoren hingegen sollen Klausuren korrigieren. Daher schreiben wir die Methode `public void KlausurKorrigieren(String modul)`:

```
package wbt03;

public class Professor extends Mensch {

	public int persnr;

	public Professor(String vorname, String nachname,
	    int geburtsjahr, int persnr) {
	      super(vorname, nachname, geburtsjahr);
	      this.persnr = persnr;
	}

	public void klausurKorrigieren(String modul) {
```

```
            System.out.println(this.vorname + " " + this.nachname
                  + " korrigiert die Klausur " + modul + ".");
      }

}
```

3.9 Objektmethoden aufrufen

Jetzt können Studierende und Professoren nicht mehr nur die Methode ihrer Superklasse Mensch aufrufen, sondern zusätzlich ihre eigene Objektmethode:

```
package wbt03;

public class Universitaet {

      public static void main(String[] args) {
            Studierender tFaber = new Studierender("Tim", "Faber",
                1995, 1234);
            tFaber.sprechen();
            tFaber.klausurSchreiben("Wirtschaftsinformatik");
      }

}
```

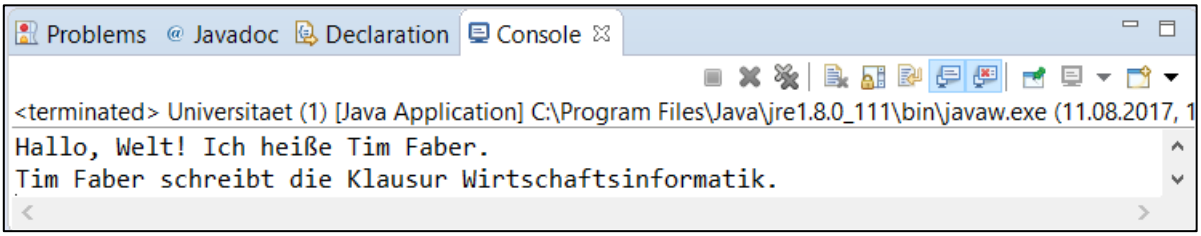


Abb. 28: Objektmethoden der Subklasse „Studierender“ ausführen

```
package wbt03;

public class Universitaet {

      public static void main(String[] args) {
            Professor cFink = new Professor("Carl", "Fink", 1965, 1);
            cFink.sprechen();
            cFink.klausurKorrigieren("Wirtschaftsinformatik");
      }

}
```

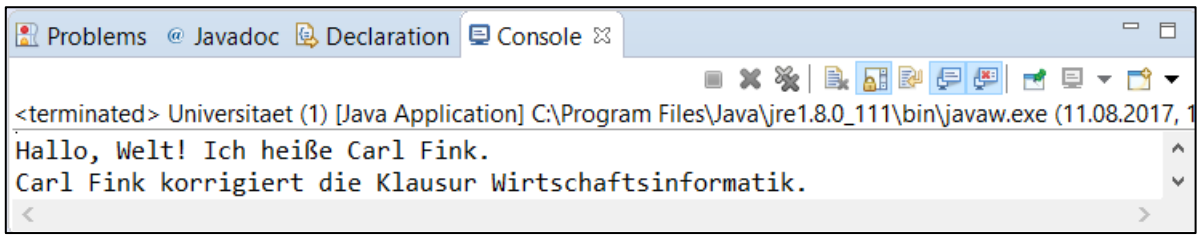


Abb. 29: Objektmethoden der Subklasse „Professor“ ausführen

Objekte der Klasse Mensch hingegen können nur ihre eigene Objektmethode aufrufen. Auf die Objektmethoden ihrer Subklassen haben Sie keinen Zugriff:

```
package wbt03;

public class Universitaet {

	public static void main(String[] args) {
		Mensch aSchmitt = new Mensch("Anna", "Schmitt", 1996);
		aSchmitt.sprechen();

		aSchmitt.klausurSchreiben("Wirtschaftsinformatik");
		aSchmitt.klausurKorrigieren("Wirtschaftsinformatik");
	}

}
```

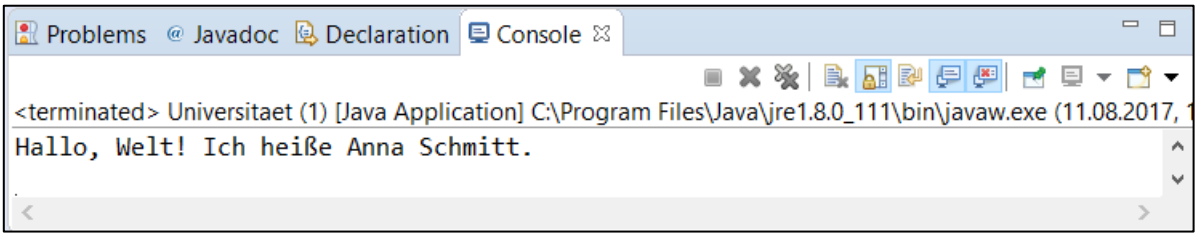


Abb. 30: Objektmethode der Superklasse „Mensch“ ausführen

3.10 Generalisierung und Spezialisierung

Sie haben in der heutigen Schulung einen Überblick über das Konzept der Vererbung erhalten. Dabei wird in diesem Zusammenhang auch häufig von Spezialisierung und Generalisierung gesprochen. Spezialisierung meint, dass eine Subklasse (hier: *Studierender* und *Professor*) das Ererbte um spezifische Eigenschaften und Verhaltensweisen erweitert. Generalisierung dagegen meint, dass gemeinsame Eigenschaften und Verhaltensweisen in einer übergeordneten Klasse zusammengefasst werden.

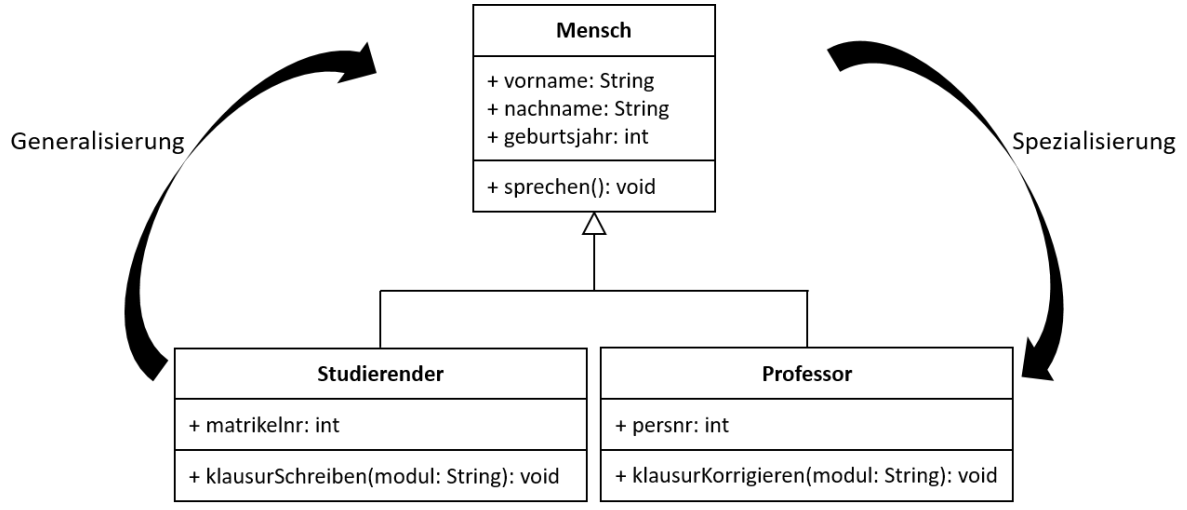


Abb. 31: Generalisierung und Spezialisierung im UML-Klassendiagramm

3.11 Abschlussaufgabe

Bitte laden Sie sich die Datei Auto.java herunter.

- **Aufgabe 1**: Überführen Sie den Quellcode der Datei Auto.java in ein UML-Klassendiagramm.
- **Aufgabe 2**: Erstellen Sie danach in Eclipse in ihrem Java-Projekt "ObjektorientiertePogrammierung" im Paket "wbt03" die Klassen *Auto* und *Fahrzeugpool* (mit main-Methode).
- **Aufgabe 3**: Bitte fügen Sie den Quellcode der Datei Auto.java in die Klasse *Auto* ein. Erzeugen Sie in der main-Methode der Klasse *Fahrzeugpool* Objekte aus der Klasse *Auto* und rufen Sie alle Objektmethoden der Klasse *Auto* auf.

3.12 Lessons learned

In der heutigen Schulung haben Sie gelernt, dass man die gemeinsamen Eigenschaften und Verhaltensweisen von ähnlichen Klassen (hier: *Studierender* und *Professor*) in einer neuen Klasse (hier: *Mensch*) zusammenfasst. Durch das Konzept der Vererbung können die sog. Subklassen (hier: *Studierender* und *Professor*) die Eigenschaften und Verhaltensweisen der sog. Superklasse (hier: *Mensch*) verwenden. Subklassen können ihre Superklasse um spezifische Eigenschaften (hier: Matrikel- und Personalnummer) und Verhaltensweise (hier: klausurSchreiben und klausurKorrigieren) erweitern.

4. Abstraktion und Überschreibung am Beispiel von Kraftfahrzeugen

4.1 Der dritte Schulungstag

Horst Schäfer: „Hallo Frau Schmitt, ich heiße Sie als Praktikantin zum dritten Tag unserer Schulung "Objektorientierte Programmierung mit Java und Eclipse" herzlich willkommen! Das Ziel der nächsten Tage ist es, mit den Konzepten der Objektorientierung, die Verwaltung einer Fahrzeugvermietung in Java zu programmieren. In der heutigen Schulung werden Sie die Konzepte der Abstraktion und Überschreibung kennenlernen.“

4.2 Das Konzept der Abstraktion

4.2.1 Vereinfachtes Beispiel

Horst Schäfer: „Sie haben bereits die Klasse *Auto* und das zugehörige UML-Klassendiagramm erstellt. In unserer Fahrzeugvermietung sollen neben Autos auch LKW angeboten werden. Bevor wir die Klasse *LKW* schreiben, müssen wir diese planen. Daher überlegen wir uns, welche Eigenschaften und Verhaltensweisen ein LKW hat.

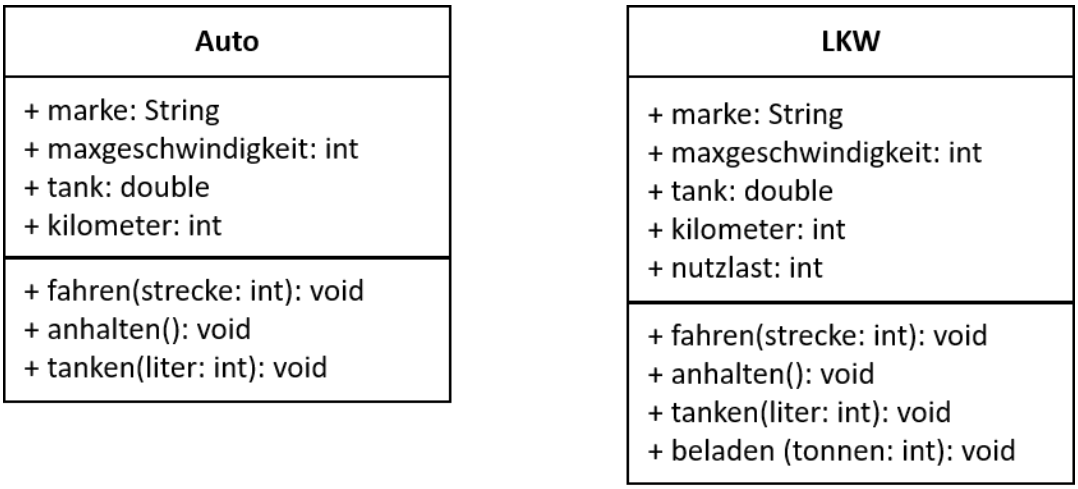


Abb. 32: UML-Klassendiagramme der Klassen „Auto“ und „LKW“

Wenn Sie die Baupläne von einem Auto und einem LKW vergleichen, werden Sie festellen, dass diese teilweise übereinstimmen. Jedoch hat hier ein LKW im Gegensatz zu einem Auto eine Nutzlast und kann beladen werden. Um die gleichen Eigenschaften und Verhaltensweisen von Autos und LKW nicht mehrfach im Quellcode schreiben zu müssen, fassen wir diese in der Klasse *Kraftfahrzeug* zusammen.“

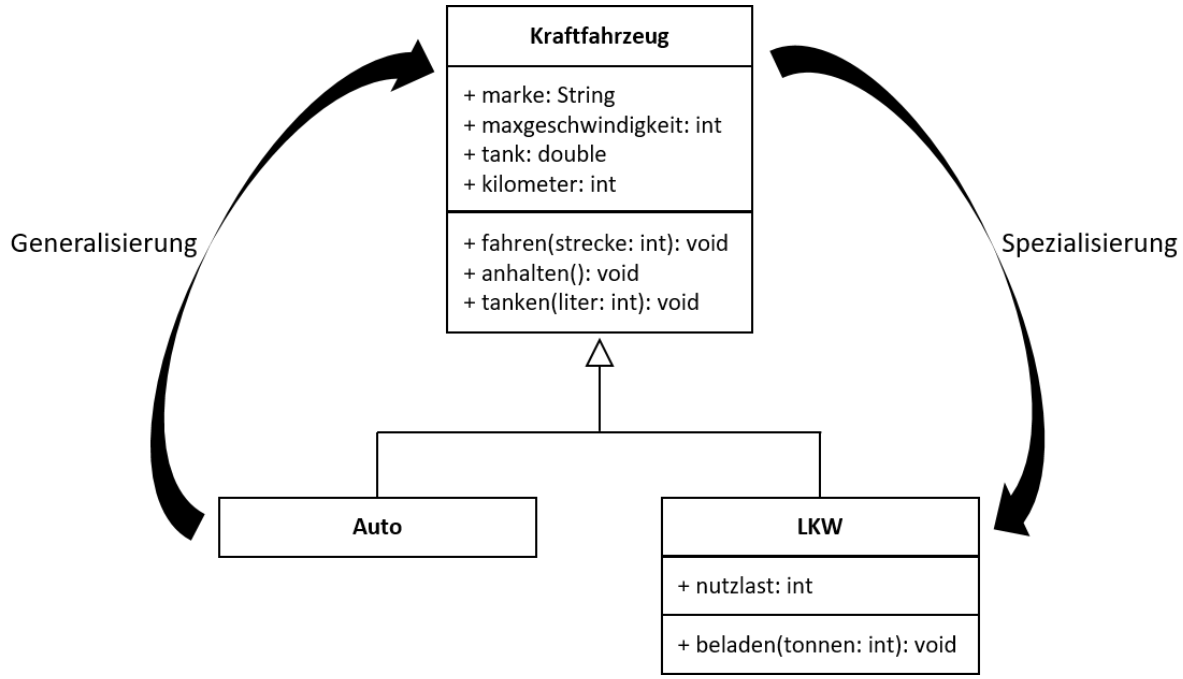


Abb. 33: Vererbung der Klasse „Kraftfahrzeug“ im UML-Klassendiagramm

4.2.2 Die Klasse Kraftfahrzeug

In unserem Java-Projekt "ObjektorientierteProgrammierung" erzeugen wir jetzt das Paket "wbt04" und erstellen nacheinander die Klassen *Kraftfahrzeug*, *Auto*, *LKW* und *Fahrzeugpool*.

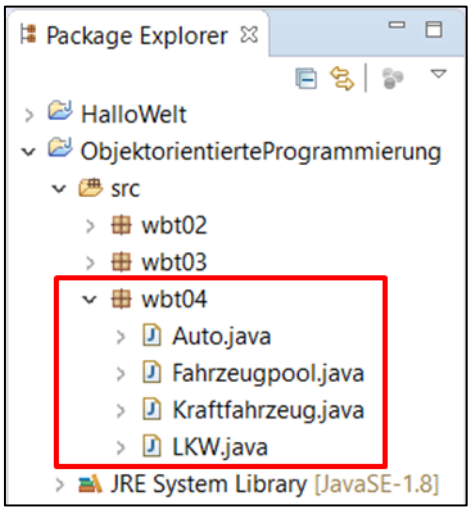


Abb. 34: Das Paket "wbt04"

Bitte schreiben Sie als erstes die Klasse Kraftfahrzeug auf Basis ihres UML-Klassendiagramms. Übertragen Sie dafür die Eigenschaften, den entsprechenden Konstruktor und die Verhaltensweisen in die Klasse *Kraftfahrzeug*:

```
package wbt04;

public class Kraftfahrzeug {

	public String marke;
	public int maxgeschwindigkeit;
	public double tank;
	public int kilometer;

	public Kraftfahrzeug(String marke, int maxgeschwindigkeit,
	    double tank, int kilometer) {
		this.marke = marke;
		this.maxgeschwindigkeit = maxgeschwindigkeit;
		this.tank = tank;
		this.kilometer = kilometer;
	}

	public void fahren(int strecke) {
		if (this.tank != 0) {
			this.kilometer += strecke;
			this.tank--;
			System.out.println("Es werden " + strecke + " Kilometer
			    gefahren.");
		} else {
			System.out.println("Ihr Tank reicht nicht aus. Bitte
			    tanken Sie!");
		}
	}

	public void tanken(int liter) {
		this.tank += liter;
		System.out.println("Der neue Tankstand beträgt: " + this.tank
		    + " Liter.");
	}

	public void anhalten() {
		System.out.println("Das Kraftfahrzeug hat angehalten.
		    Kilometerstand: " + this.kilometer + " Kilometer.
		    Tankstand: " + this.tank + " Liter.");
	}

}
```

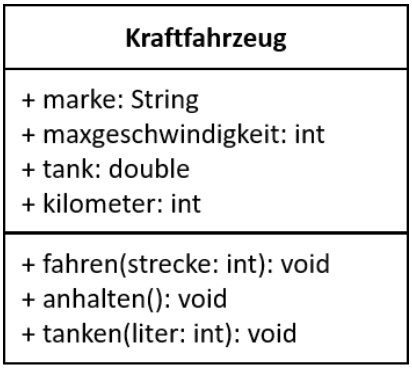


Abb. 35: UML-Klassendiagramm der Klasse „Kraftfahrzeug“

4.2.3 Die Klassen Auto und LKW

Jetzt wollen wir die Klassen *Auto* und *LKW* schreiben. Da Autos und LKW Ausprägungen eines Kraftfahrzeugs sind, sollen sie von der Klasse Kraftfahrzeug erben. Dafür verwenden wir wieder das Schlüsselwort extends. Zusätzlich benötigen beide Klassen einen Konstruktor. Dieser greift mit dem Schlüsselwort super auf den Konstruktor der Superklasse zu:

```
package wbt04;

public class Auto extends Kraftfahrzeug {

	public Auto(String marke, int maxgeschwindigkeit, double tank,
	  int kilometer) {
	    super(marke, maxgeschwindigkeit, tank, kilometer);
	}

}
```

```
package wbt04;

public class LKW extends Kraftfahrzeug {

	public int nutzlast;

	public LKW(String marke, int maxgeschwindigkeit, double tank, int kilometer,
	  int nutzlast) {
	    super(marke, maxgeschwindigkeit, tank, kilometer);
	    this.nutzlast = nutzlast;
	}

	public void beladen(int tonnen) {
		if (this.nutzlast >= tonnen) {
			System.out.println("Der LKW wurde mit " + tonnen
			  + " Tonnen beladen.");
		} else
			System.out.println("Die Ladung ist zu schwer.");
	}

}
```

4.2.4 Kraftfahrzeuge, Autos und LKW erzeugen

Nachdem wir den Konstruktor für Autos und LKW geschrieben haben, können wir aus den Klassen *Kraftfahrzeug*, *Auto* und *LKW* Objekte erzeugen. Dabei können Autos und LKW mit der Punktnotation auch auf die Methoden der Klasse *Kraftfahrzeug* zugreifen, da die Klassen *Auto* und *LKW* von der Klasse *Kraftfahrzeug* erben.

Wir erzeugen in der Klasse *Fahrzeugpool* die folgenden Objekte:

- "audi": Kraftfahrzeug, marke = audi, maxgeschwindigkeit = 200, tank = 1, kilometer = 150:

```
package wbt04;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Kraftfahrzeug audi = new Kraftfahrzeug("Audi", 200, 1,
			150);
	}

}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Fahrzeugpool (1) [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (21.08.2017, 09:19:
Der neue Tankstand beträgt 41.0 Liter.
Es werden 30 Kilometer gefahren.
Das Kraftfahrzeug hat angehalten. Kilometerstand: 180 Kilometer. Tankstand: 40.0 Liter.
```

Abb. 36: Objektmethoden der Klasse „Kraftfahrzeug“ ausführen

- "ferrari": Auto, marke = Ferrari, maxgeschwindigkeit = 350, tank = 0, kilometer = 0:

```
package wbt04;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto ferrari = new Auto("Ferrari", 350, 0, 0);
		ferrari.tanken(50);
		ferrari.fahren(45);
		ferrari.anhalten();
	}

}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Fahrzeugpool (1) [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (21.08.2017, 09:19:
Der neue Tankstand beträgt 50.0 Liter.
Es werden 45 Kilometer gefahren.
Das Kraftfahrzeug hat angehalten. Kilometerstand: 45 Kilometer. Tankstand: 49.0 Liter.
```

Abb. 37: Objektmethoden der Klasse „Auto“ ausführen

- "man": LKW, marke = MAN, maxgeschwindigkeit = 70, tank = 40, kilometer = 200, nutzlast = 120:

```
package wbt04;

public class Fahrzeugpool {

	public static void main(String[] args) {
		LKW man = new LKW("MAN", 70, 40, 200, 120);
		man.beladen(100);
		man.tanken(50);
		man.fahren(45);
		man.anhalten();
```

```
        }

}
```

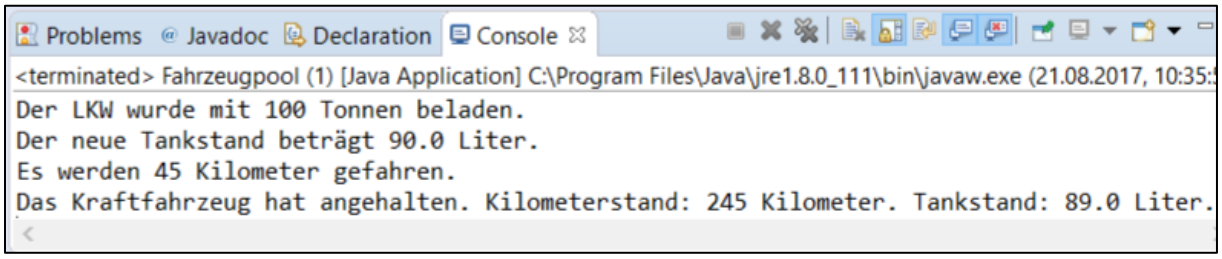


Abb. 38: Objektmethoden der Klasse „LKW“ ausfüren

4.2.5 Abstrakte Klassen

Bisher können wir Objekte aus den Klassen *Kraftfahrzeug*, *Auto* und *LKW* erzeugen. In der Realität existieren Kraftfahrzeuge jedoch nur in speziellen Ausprägungen. Zum Beispiel als Autos und LKW. Wir wollen ab jetzt verhindern, dass aus der Klasse *Kraftfahrzeug* Objekte erzeugt werden können. Die Klasse *Kraftfahrzeug* soll uns nur als Bauplan für Autos und LKW dienen. Dafür verwenden wir das Schlüsselwort abstract:

```
package wbt04;

public abstract class Kraftfahrzeug {

      // Eigenschaften //Konstruktor // Methoden

}
```

Mit abstract kann die Klasse *Kraftfahrzeug* ihre Eigenschaften und Verhaltensweisen weiterhin vererben, aber ein Objekt kann aus der Klasse *Kraftfahrzeug* nicht mehr erzeugt werden:

```
package wbt04;

public class Fahrzeugpool {

      public static void main(String[] args) {
            Kraftfahrzeug audi = new Kraftfahrzeug("Audi", 200, 1, 150);
            Auto ferrari = new Auto("Ferrari", 350, 0, 0);
            LKW man = new LKW("MAN", 70, 40, 200, 120);
      }

}
```

Obwohl aus abstrakten Klassen keine Objekte erzeugt werden können, haben sie einen Konstruktor. Der Konstruktor einer abstrakten Klasse kann zwar nicht mit dem new-Befehl aufgerufen werden, jedoch können Subklassen mit dem super-Befehl auf den Konstruktor ihrer abstrakten Superklasse zugreifen.

4.2.6 Abstrakte Klassen im UML-Klassendiagramm

Im UML-Klassendiagramm wird der Name einer abstrakten Klasse kursiv dargestellt. Dabei können abstrakte Klassen auch sog. abstrakte Methoden enthalten.

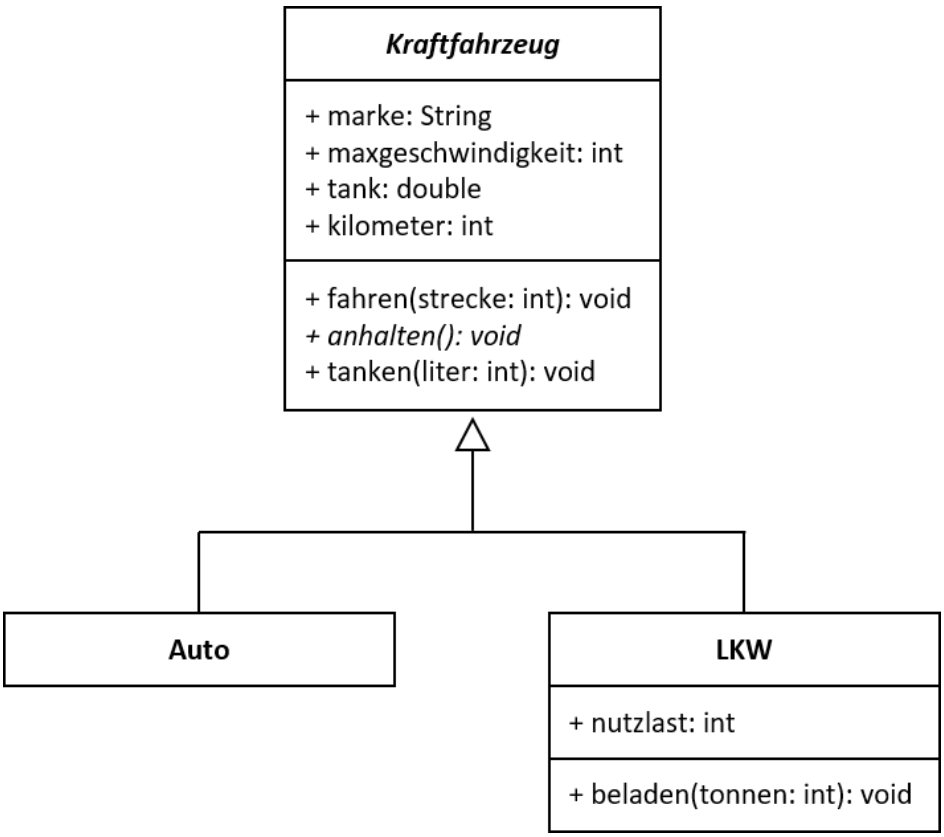


Abb. 39: Abstrakte Klasse „Kraftfahrzeug“ im UML-Klassendiagramm

Abstrakte Methoden (hier: `public abstract void anhalten()`) werden im UML-Klassendiagramm kursiv dargestellt. Was eine abstrakte Methode ist und wann sie verwendet wird, erfahren Sie auf den nächsten Seiten.

4.2.7 Abstrakte Methoden

Jede Klasse, die eine abstrakte Methode enthält, ist automatisch eine abstrakte Klasse. Aus dem UML-Klassendiagramm der Klasse Kraftfahrzeug können wir ablesen, dass in der Klasse *Kraftfahrzeug* die abstrakte Methode `public void anhalten()` steht. Mit dieser Methode schreibt die Klasse *Kraftfahrzeug* ihren Subklassen verbindlich vor, dass sie die Verhaltensweise "anhalten" in ihrem Bauplan definieren müssen. Dem Programmierer der Subklasse wird überlassen, wie er die anhalten()-Methode genau gestaltet. So können wir unterschiedliche Befehle in den anhalten()-Methoden der Klassen *Auto* und *LKW* definieren.

4.2.8 Abstrakte Methoden in Java

Eine abstrakte Methode wird genauso wie eine abstrakte Klasse mit dem Schlüsselwort abstract gekennzeichnet:

```
package wbt04;

public abstract class Kraftfahrzeug {

	// Eigenschaften // Konstruktor // Methoden

	public abstract void anhalten();

}
```

Im Gegensatz zu nicht-abstrakten Methoden, haben abstrakte Methoden keinen Methodenblock, denn sie beinhalten keine Befehle. Erst in den Subklassen *Auto* und *LKW* werden in der anhalten()-Methode die entsprechenden Befehle definiert:

```
package wbt04;

public class Auto extends Kraftfahrzeug {

	// Konstruktor

	public void anhalten() {
		System.out.println("Das Auto hat angehalten. Kilometerstand: "
		  + this.kilometer + " Kilometer. Tankstand: " + this.tank
		  + " Liter.");
	}

}
```

```
package wbt04;

public class LKW extends Kraftfahrzeug {

	// Eigenschaft // Konstruktor // Methode

	public void anhalten() {
		System.out.println("Der LKW hat angehalten. Kilometerstand: "
		  + this.kilometer + " Kilometer. Tankstand: " + this.tank
		  + " Liter.");
	}

}
```

4.3 Das Konzept der Überschreibung

4.3.1 Überschreibung

Java bietet Programmierern die Möglichkeit das Konzept der Überschreibung zu benutzen. Dieses haben wir bereits auf der vorherigen Seite angewandt, indem wir die Methode `public void anhalten()` in die Klassen *Auto* und *LKW* geschrieben haben. Überschreibung bedeutet, dass eine Subklasse (hier: *Auto*, *LKW*) eine Methode besitzt, die bereits in einer übergeordneten Klasse (hier: *Kraftfahrzeug*) deklariert bzw. definiert wurde.

```
package wbt04;
public abstract class Kraftfahrzeug {
   public abstract void anhalten();
}
```

```
package wbt04;
public class Auto extends Kraftfahrzeug {
   public void anhalten() { ... }
}
```

```
package wbt04;
public class LKW extends Kraftfahrzeug {
   public void anhalten() { ... }
}
```

Abb. 40: Überschreibung der Methode „anhalten“

4.3.2 Aufruf der überschriebenen Methoden

Nachdem wir die anhalten()-Methode in den Subklassen überschrieben haben, ist die Vererbung der abstrakten Klasse *Kraftfahrzeug* an die Klassen *Auto* und *LKW* abgeschlossen. Wenn wir jetzt die Klasse *Fahrzeugpool* erneut ausführen, erhalten wir in der Konsole die angepassten Ausgaben der Klassen *Auto* und *LKW*:

```
package wbt04;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto ferrari = new Auto("Ferrari", 350, 0, 0);
		LKW man = new LKW("MAN", 70, 40, 200, 120);

		ferrari.tanken(50);
		ferrari.fahren(45);
		ferrari.anhalten();

		man.beladen(100);
		man.tanken(50);
		man.fahren(45);
		man.anhalten();
	}

}
```

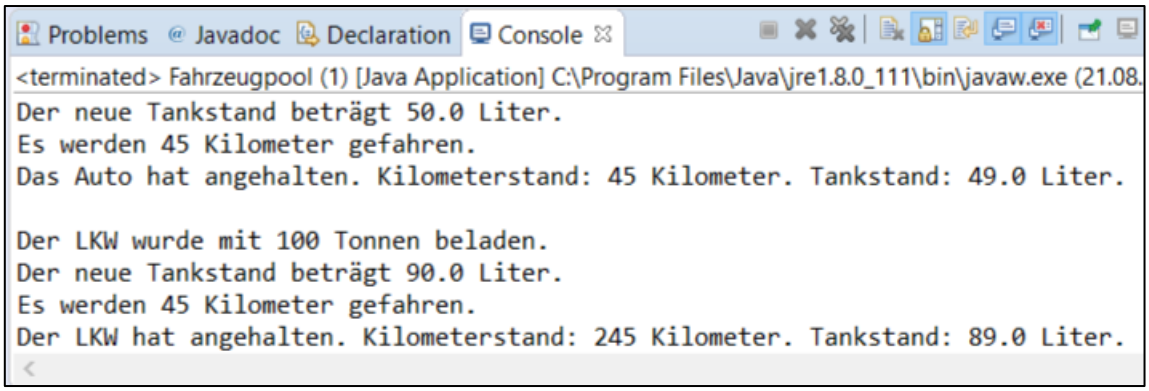


Abb. 41: Überschriebene Methode „anhalten“ ausführen

4.3.3 Die Annotation @Override

Nachdem wir die anhalten()-Methode eines Autos und eines LKW aufgerufen haben, schauen wir uns deren Baupläne noch einmal an. Dabei können wir ohne die Klasse *Kraftfahrzeug* zu kennen, nicht wissen, ob die anhalten()-Methode eine eigene Mehode der Subklassen ist oder ob sie von der Superklasse geerbt und überschrieben wurde. Daher schreiben wir in den Klassen *Auto* und *LKW* vor die anhalten()-Methode die Annotation @Override, um zu kennzeichnen, dass die anhalten()-Methode der Klasse *Kraftfahrzeug* überschrieben wird:

```
package wbt04;

public class Auto extends Kraftfahrzeug {

	// Konstruktor

	@Override
	public void anhalten() { ... }

}
```

```
package wbt04;

public class LKW extends Kraftfahrzeug {

	// Eigenschaft // Konstruktor // Methode

	@Override
	public void anhalten() { ... }


}
```

Wenn wir die Annotation @Override vor eine überschriebene Methode in einer Subklasse schreiben, überprüft der Compiler beim Ausführen des Programmes, ob die Superklasse diese Methode tatsächlich enthält und gibt einen Fehler aus, wenn nicht.

4.3.4 Exkurs: Vererbung abstrakter Methoden

Wenn wir die abstrakte anhalten()-Methode in den Subklassen *Auto* und *LKW* nicht überschreiben, wird sie als abstrakte Methode vererbt. Da jede Klasse, die eine abstrakte Methode enthält, selbst zu einer abtrakten Klasse wird, können wir diese nicht mehr zur Erzeugung eines Objekts verwenden. Das heißt, wir könnten weder aus der Klasse *Kraftfahrzeug*, noch aus den Klassen *Auto* und *LKW* Objekte erzeugen:

```
public class Fahrzeugpool {

	public static void main(String[] args) {
		Kraftfahrzeug audi = new Kraftfahrzeug("Audi", 200, 1, 150);
		Auto ferrari = new Auto("Ferrari", 350, 0, 0);
		LKW man = new LKW("MAN", 70, 40, 200, 120);
	}

}
```

4.3.5 Lessons learned

Am heutigen Schulungstag haben Sie gelernt, dass abstrakte Klassen

- in der Klassendefinition mit dem Schlüsselwort abstract gekennzeichnet werden,
- selbst keine Objekte erzeugen können und
- abstrakte Methoden enthalten können, aber nicht müssen.

Zudem müssen Sie beim Vererben von abstrakten Klassen sicherstellen, dass Sie alle abstrakten Methoden überschreiben, wenn Sie aus der Subklasse Objekte erzeugen wollen.

5. Überladung und Polymorphie am Beispiel von Autos und LKW

5.1 Der vierte Schulungstag

Horst Schäfer: „Hallo Frau Schmitt, ich heiße Sie als Praktikantin zum vierten Tag unserer Schulung "Objektorientierte Programmierung mit Java und Eclipse" herzlich willkommen! Gestern haben Sie bereits die Konzepte der Abstraktion und Überschreibung am Beispiel einer Autovermietung kennengelernt. Dieses Beispiel wollen wir auch in der heutigen Schulung für das Konzept der Polymorphie heranziehen.“

5.2 Das Paket "wbt05" erzeugen

Heute wollen wir an den Klassen *Auto*, *Fahrzeugpool*, *Kraftfahrzeug* und *LKW* weiterarbeiten, die wir am letzten Schulungstag im Paket "wbt04" erzeugt haben. Um die Klassen nicht erneut schreiben zu müssen, wollen wir sie kopieren.

Bitte führen Sie dafür die folgenden Schritte durch:

1. Bitte erzeugen Sie zuerst das Paket "wbt05" im Java-Projekt "ObjektorientierteProgrammierung".
2. Anschließend kopieren Sie die Klassen *Auto*, *Fahrzeugpool*, *Kraftfahrzeug* und *LKW* im Paket "wbt04". Bitte markieren Sie in Ihrem Package Explorer im Paket "wbt04" die Klassen *Auto*, *Fahrzeugpool*, *Kraftfahrzeug* und *LKW*. Klicken Sie danach mit der rechten Maustaste auf die markierten Klassen und wählen Sie den Menüpunkt "copy".
3. Jetzt können Sie die Klassen in das Paket "wbt05" einfügen. Klicken Sie in Ihrem Package Explorer mit der rechten Maustaste auf das Paket "wbt05", in das Sie die Klassen einfügen wollen und wählen Sie den Menüpunkt "paste".

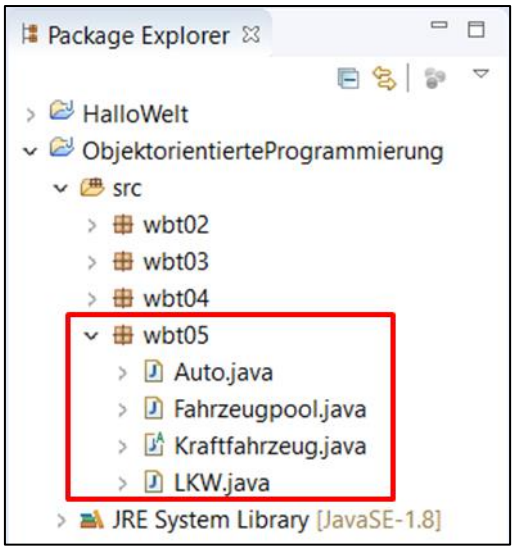


Abb. 42: Das Paket „wbt05“

5.3 Die Klasse Object

Horst Schäfer: „Das Konzept der Vererbung haben Sie bereits kennengelernt. Wir haben jedoch noch nicht darüber gesprochen, dass in Java alle Klassen eine gemeinsame Superklasse haben. Die Klasse Object.

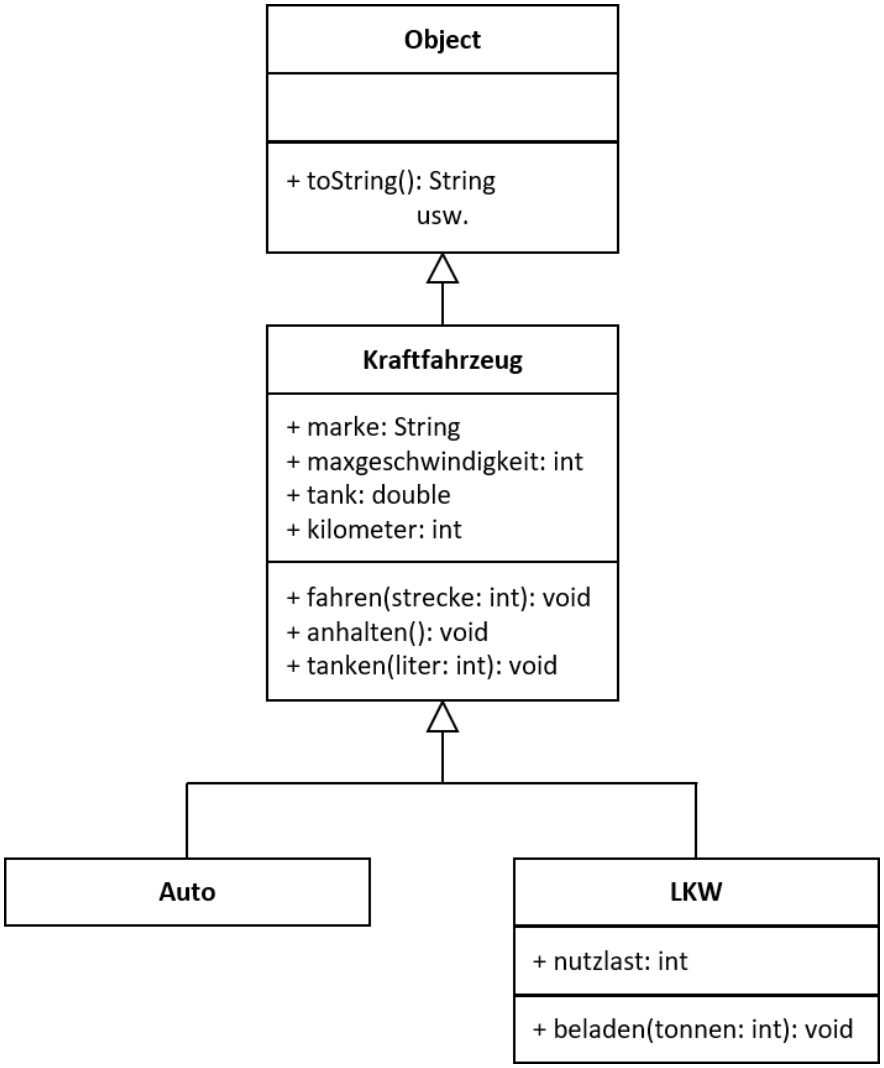


Abb. 43: Die Klasse „Object“ im UML-Klassendiagramm

In der Klasse Object stehen mehrere Methoden. Eine dieser Methoden ist `public String toString()`. Das Ziel der toString()-Methode ist es, ein Objekt in Textform auszugeben. Die Ausgabe funktioniert standardmäßig jedoch nicht wie gewünscht. Bitte geben Sie in die main-Methode der Klasse Fahrzeugpool den Befehl `System.out.println(ferrari.toString());` ein, um dies zu demonstrieren:“

```
package wbt05;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto ferrari = new Auto("Ferrari", 350, 0, 0);
		System.out.println(ferrari.toString());
	}


}
```

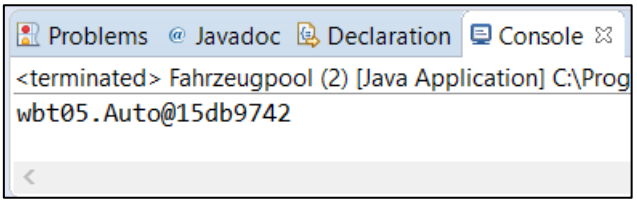


Abb. 44: Die Methode „toString“ der Klasse „Object“ ausführen

5.4 Die toString()-Methode

Wie Sie auf der vorherigen Seite gesehen haben, liefert der Aufruf der toString()-Methode der Klasse *Object* keine aussagekräftige Bildschirmausgabe. Mithilfe der Überschreibung können wir die toString()-Methode jedoch so anpassen, dass wir sie sinnvoll nutzen können. Sie soll alle Werte der Eigenschaften eines Kraftfahrzeugs in der Konsole ausgeben. Dafür überschreiben wir als erstes die toString()-Methode in der Klasse *Kraftfahrzeug*:

```
package wbt05;

public abstract class Kraftfahrzeug {

	// Eigenschaften // Konstruktor // Methoden

	@Override
	public String toString() {
		return "Marke=" + this.marke + ", Hoechstgeschwindigkeit="
			+ this.maxgeschwindigkeit + ", Tankstand=" + this.tank
			+ ", Kilometerstand=" + this.kilometer;
	}


}
```

Danach überschreiben wir die toString()-Methode auch in der Klasse *LKW*. Rufen Sie beim Überschreiben der toString()-Methode in der Klasse *LKW* mit dem super-Befehl die toString()-Methode der Klasse *Kraftfahrzeug* auf und fügen Sie die Ausgabe der Nutzlast hinzu:

```
package wbt05;

public class LKW extends Kraftfahrzeug {

	// Eigenschaft // Konstruktor // Methoden

	@Override
	public String toString() {
		return super.toString() + ", Nutzlast=" + this.nutzlast;
	}

}
```

5.5 Aufruf der toString()-Methode

Nachdem wir die Methode `public String toString()` in den Klassen *Kraftfahrzeug* und *LKW* überschrieben haben, erzeugen wir Objekte aus den Klassen *Auto* und *LKW*. Anschließend rufen wir mit den beiden Objekten die toString()-Methode im Ausgabebefehl auf. Jetzt werden uns die soeben definierten Ausgaben der Klassen Auto und LKW angezeigt:

```
package wbt05;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto ferrari = new Auto("Ferrari", 350, 0, 0);
		LKW man = new LKW("MAN", 70, 40, 200, 120);

		System.out.println(ferrari.toString());
		System.out.println(man.toString());
	}
}
```

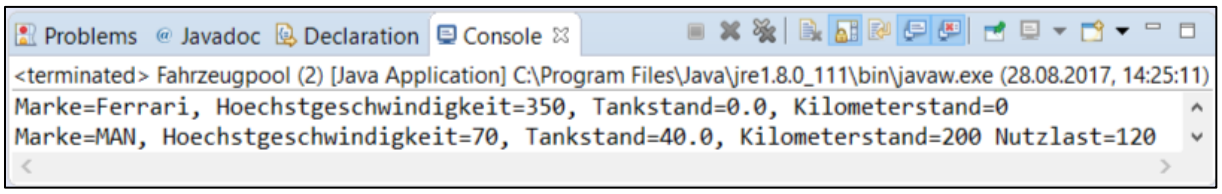


Abb. 45: Überschriebene Methode „toString“ ausführen

5.6 Überschreiben der toString()-Methode

Sie haben gerade die Methode `public String toString()` der Klasse *Object* kennengelernt. Deren Ziel ist es, ein Objekt in Textform auszugeben. Jedoch liefert die Methode beim Aufruf nur eine unleserliche Zeichenkette. Daher haben wir die toString()-Methode so überschrieben, dass sie uns alle Eigenschaften eines Kraftfahrzeugs in der Konsole anzeigt.

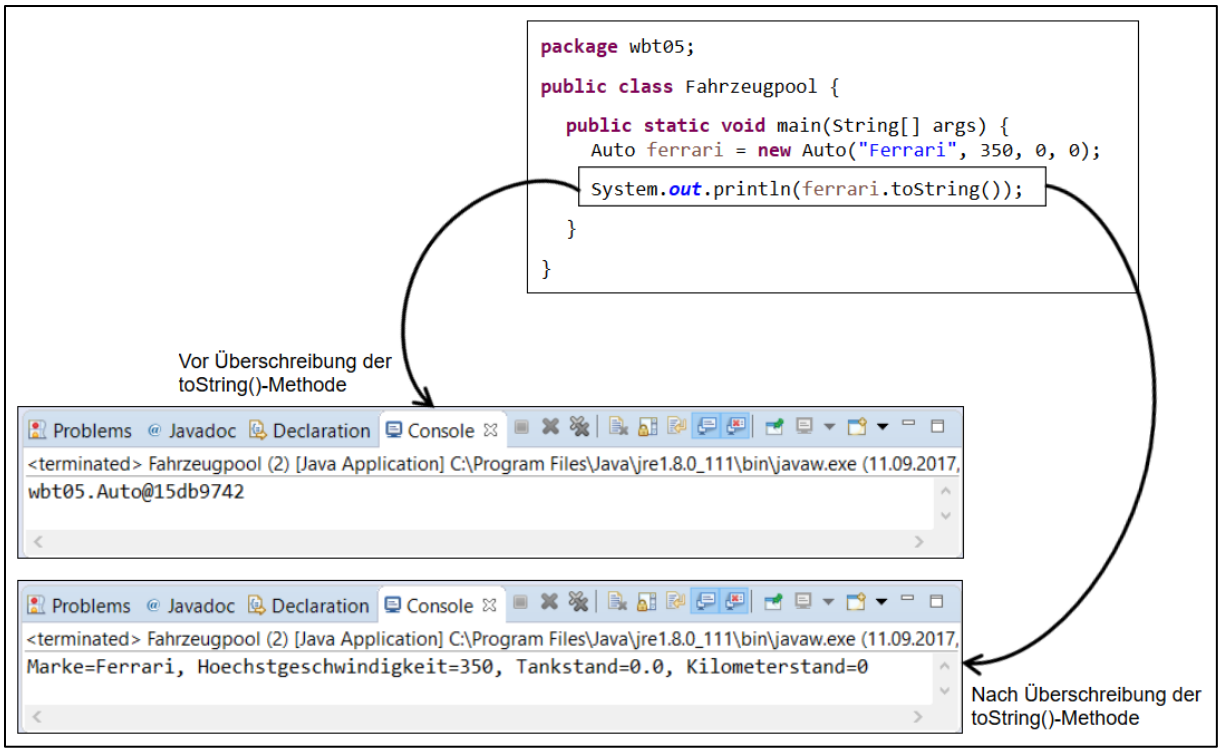


Abb. 46: Die Methode „toString“ vor und nach Überschreibung

5.7 Überladung

Horst Schäfer: „Wie Sie bereits gelernt haben, können wir Methoden, z. B. die toString()-Methode, überschreiben. Das bedeutet, wir schreiben eine Methode in eine Subklasse, obwohl sie bereits in einer übergeordneten Klasse deklariert bzw. definiert wurde. Wir können Methoden aber nicht nur überschreiben, sondern auch überladen. Überladen bedeutet, dass wir in eine Klasse mehrere Methoden mit dem gleichen Namen schreiben. Sie unterscheiden sich jedoch durch ihre Parameter und Befehle. Obwohl mehrere Methoden den gleichen Namen haben, ruft der Compiler automatisch die richtige Methode auf. Welche Methode aufgerufen werden soll, erkennt der Compiler durch die Klassenzugehörigkeit des Objekts und anhand der übergebenen Parameter.“

5.8 Überladen von Methoden in Java

In die Klasse *Kraftfahrzeug* fügen wir eine zweite Methode `public void fahren` hinzu. Diese zweite Methode berücksichtigt neben der Strecke auch den Verbrauch. Wir übergeben also einen zusätzlichen Parameter (`double verbrauch`). Dieser wird in den Befehlen benötigt, um den Tankstand in Abhängigkeit von der gefahrenen Strecke und dem Verbrauch pro Kilometer zu reduzieren:

```
package wbt05;

public abstract class Kraftfahrzeug {

	// Eigenschaften // Konstruktor // Methoden

	public void fahren(int strecke) {
		if (this.tank != 0) {
			this.kilometer += strecke;
			this.tank--;
			System.out.println("Es werden " + strecke
			  + " Kilometer gefahren.");
		} else
			System.out.println("Ihr Tank reicht nicht aus.
			  Bitte tanken Sie!");
	}

	// ueberladene Methode
	public void fahren(int strecke, double verbrauch) {
		if (this.tank >= strecke * verbrauch) {
			this.kilometer += strecke;
			this.tank -= strecke * verbrauch;
			System.out.println("Es werden " + strecke
			+ " Kilometer gefahren.");
		} else {
			System.out.println("Ihr Tank reicht nicht aus.
			Bitte tanken Sie!");
		}

	}

}
```

5.9 Aufruf der überladenen Methode

Nachdem wir die Methode `public void fahren` in der Klasse *Kraftfahrzeug* überladen haben, wollen wir die beiden fahren()-Methoden in der Klasse *Fahrzeugpool* testen. Dafür erzeugen wir das Objekt "ferrari" der Klasse *Auto*. Zuerst rufen wir die ursprüngliche fahren()-Methode auf und übergeben einen Parameter. Wir fahren 45 Kilometer. Danach rufen wir die überladene fahren()-Methode erneut auf und übergeben zwei Parameter. Wir fahren 45 Kilometer und haben einen verbrauchen 0,12 Liter pro Kilometer:

```
package wbt05;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto ferrari = new Auto("Ferrari", 350, 0, 0);
		LKW man = new LKW("MAN", 70, 40, 200, 120);

		ferrari.tanken(50);
```

```
            ferrari.fahren(45);
            ferrari.anhalten();

            ferrari.fahren(45, 0.12);
            ferrari.anhalten();
      }

}
```

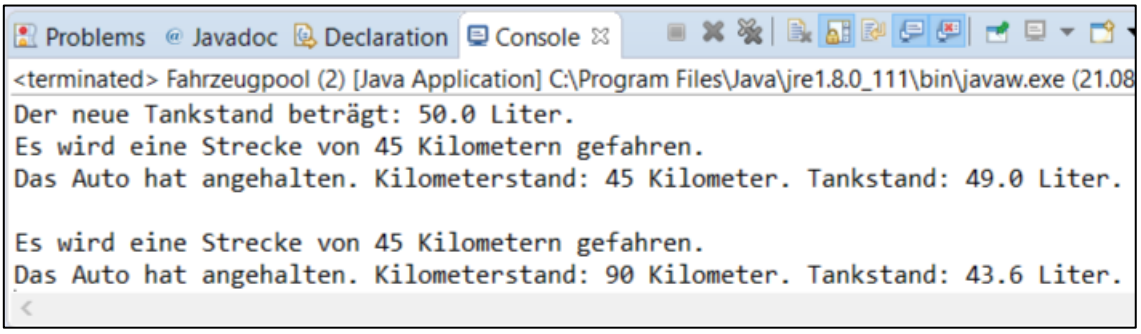


Abb. 47: Überladene Methode „fahren“ ausführen

5.10 Überladen von Konstruktoren in Java

In Java können wir aber nicht nur Methoden, sondern auch Konstruktoren überladen. Wir wollen mithilfe der Überladung in unserer Fahrzeugvermietung auch Neuwagen anbieten. Der Tank- und Kilometerstand soll mit 0 angezeigt werden. Wir schreiben einen zweiten Konstruktor in die Klasse *Kraftfahrzeug*, der lediglich die Parameter `marke` und `maxgeschwindigkeit` berücksichtigt. Im Methodenblock dieses Konstruktors weisen wir den Eigenschaften Tank und Kilometer den Wert 0 zu, denn es handelt sich um Neuwagen:

```
package wbt05;

public abstract class Kraftfahrzeug {

      public String marke; // Marke
      public int maxgeschwindigkeit; // Hoechstgeschwindigkeit
      public double tank; // Tankstand
      public int kilometer; // Kilometerstand

      public Kraftfahrzeug(String marke, int maxgeschwindigkeit,
                  double tank, int kilometer) {
            this.marke = marke;
            this.maxgeschwindigkeit = maxgeschwindigkeit;
            this.tank = tank;
            this.kilometer = kilometer;
      }

      // ueberladener Konstruktor
      public Kraftfahrzeug(String marke, int maxgeschwindigkeit) {
            this.marke = marke;
            this.maxgeschwindigkeit = maxgeschwindigkeit;
            this.tank = 0;
```

```
            this.kilometer = 0;
      }

}
```

Da *Kraftfahrzeug* eine abstrakte Klasse ist, können wir aus ihr keine Objekte erzeugen. Bevor wir also den überladenen Konstruktor aufrufen können, müssen wir auch die Konstruktoren der Klassen *Auto* und *LKW* überschreiben. Dafür schreiben wir jeweils einen zweiten Konstruktor in die Klassen *Auto* und *LKW* und rufen im Methodenblock mit dem super()-Befehl den überladenen Konstruktor der Superklasse *Kraftfahrzeug* auf:

```
package wbt05;

public class Auto extends Kraftfahrzeug {

      public Auto(String marke, int maxgeschwindigkeit, double tank,
        int kilometer) {
            super(marke, maxgeschwindigkeit, tank, kilometer);
      }

      // ueberladener Konstruktor
      public Auto(String marke, int maxgeschwindigkeit) {
            super(marke, maxgeschwindigkeit);
      }

}
```

Wir dürfen nicht vergessen, dass LKW die spezialisierte Eigenschaft Nutzlast haben. Daher müssen wir zusätzlich auch den Parameter Nutzlast im überladenen Konstruktor übergeben und zuweisen:

```
package wbt05;

public class LKW extends Kraftfahrzeug {

      // Eigenschaft // Methoden

      public LKW(String marke, int maxgeschwindigkeit, double tank,
              int kilometer, int nutzlast) {
            super(marke, maxgeschwindigkeit, tank, kilometer);
            this.nutzlast = nutzlast;
      }

      // ueberladener Konstruktor
      public LKW(String marke, int maxgeschwindigkeit, int nutzlast) {
            super(marke, maxgeschwindigkeit);
            this.nutzlast = nutzlast;
      }

}
```

5.11 Aufruf der überladenen Konstruktoren

Nachdem wir die Konstruktoren der Klassen *Auto* und *LKW* überladen haben, erzeugen wir in der Klasse *Fahrzeugpool* Neuwagen aus den Klassen *Auto* und *LKW*:

- "lamborghini": Auto, marke = lamborghini, maxgeschwindigkeit = 400
- "scania": Auto, marke = Scania, maxgeschwindigkeit: 80, Nutzlast: 30

Danach lassen wir uns die Eigenschaften der erzeugten Objekte durch den Aufruf der toString()-Methode ausgeben:

```
package wbt05;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto lamborghini = new Auto("Lamborghini", 400);
		LKW scania = new LKW("Scania", 80, 30);

		System.out.println(lamborghini.toString());
		System.out.println(scania.toString());
	}

}
```

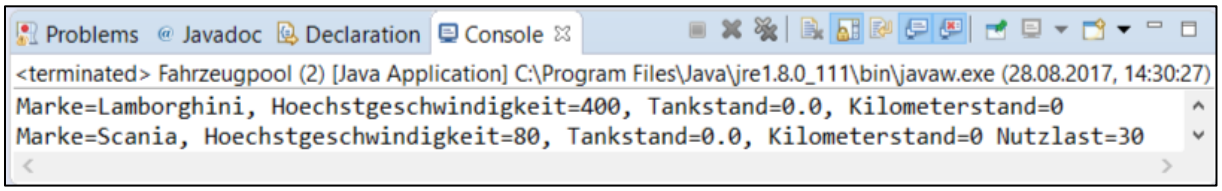


Abb. 48: Überladene Konstruktoren der Klassen „Auto“ und „LKW“ ausführen

5.12 Zum Begriff “Polymorphie“

Anna Schmitt: „Mit der Überschreibung und Überladung haben Sie uns gezeigt, dass in Java eine Methode in verschiedenen Gestalten auftreten kann. Sie kann in einer oder mehreren Klassen, mit verschiedenen Befehlen und unterschiedlichen Parametern stehen.“

Horst Schäfer: „Genau! In diesem Zusammenhang spricht man auch von "Polymorphie". Wörtlich übersetzt, bedeutet Polymorphie Vielgestaltigkeit. Eine Methode kann also in vielen verschiedenen Gestalten in einer oder verschiedenen Klassen auftreten. Durch die Klassenzugehörigkeit des Objekts und anhand der übergebenen Parameter, ruft der Compiler automatisch die richtige Methode auf.“

5.13 Lessons learned

In der heutigen Schulung haben Sie gelernt, dass in Java Methoden in unterschiedlichen Gestalten auftreten können:

- **Überschreibung**: Durch Überschreibung wird eine von der Superklasse geerbte Methode durch eine eigene Methode in einer Subklasse ersetzt.
- **Überladung**: Es gibt in einer Klasse mehrere Methoden mit dem gleichen Namen, die sich nur im Typ oder der Anzahl ihrer Parameter unterscheiden.

Im Gegensatz zur Überschreibung, ist bei der Überladung keine Vererbungsbeziehung erforderlich. Zudem können wir überladene und überschriebene Methoden in Java durch die Annotation @Override unterscheiden.

6. Autos und LKW verbergen durch Kapselung ihre Eigenschaften

6.1 Der fünfte Schulungstag

Horst Schäfer: „Hallo Frau Schmitt, ich heiße Sie als Praktikantin zum fünften Tag unserer Schulung "Objektorientierte Programmierung mit Java und Eclipse" herzlich willkommen! In den vergangenen Tagen haben Sie bereits die Konzepte Abstraktion, Überschreibung und Polymorphie am Beispiel einer Autovermietung kennengelernt. Dieses Beispiel werden wir auch in der heutigen Schulung für das Konzept der Kapselung heranziehen.“

6.2 Das Paket “wbt06“ erzeugen

Heute wollen wir an den Klassen *Auto*, *Fahrzeugpool*, *Kraftfahrzeug* und *LKW* weiterarbeiten. Diese haben wir am letzten Schulungstag im Paket "wbt05" erstellt. Bitte erzeugen Sie ein neues Paket "wbt06" in Ihrem Java-Projekt "ObjektorientierteProgrammierung". Kopieren Sie danach die Klassen *Auto*, *Fahrzeugpool*, *Kraftfahrzeug* und *LKW* im Paket "wbt05" und fügen Sie diese in das Paket "wbt06" ein.

6.3 Kapselung

Horst Schäfer: „Bis jetzt können wir die Eigenschaften von Autos, LKW und Kraftfahrzeugen beliebig manipulieren. Zum Beispiel kann die Höchstgeschwindigkeit eines Autos oder LKW nach der Erzeugung auf 1.000.000 km/h gesetzt werden:

```
package wbt06;

public class Fahrzeugpool {

     public static void main(String[] args) {
           Auto lamborghini = new Auto("Lamborghini", 400);
           LKW scania = new LKW("Scania", 80, 30);

           lamborghini.maxgeschwindigkeit = 1000000;
           scania.maxgeschwindigkeit = 1000000;
           System.out.println(lamborghini.toString());
           System.out.println(scania.toString());
     }

}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Fahrzeugpool (3) [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (11.09.2017, 13:06:36)
Marke=Lamborghini, Hoechstgeschwindigkeit=1000000, Tankstand=0.0, Kilometerstand=0
Marke=Scania, Hoechstgeschwindigkeit=1000000, Tankstand=0.0, Kilometerstand=0, Nutzlast=30
```

Abb. 49: Eigenschaften mit einem direkten Zugriff manipulieren

Um ab jetzt zu verhindern, dass die Eigenschaften solch ungewünschte Werte annehmen, werden wir diese kapseln. Nach der Kapselung kann nur noch die Klasse selbst diekt auf ihre Eigenschaften zugreifen. Alle anderen Klassen und Subklassen haben auf die Eigenschaften keinen direkten Zugriff mehr. So können wir verhindern, dass die Eigenschaften ungewünschte Werte annehmen.“

6.4 Kapselung in Java

Bisher sind alle Eigenschaften in den Klassen *Kraftfahrzeug* und *LKW* als `public` gekennzeichnet. Das heißt, jede andere Klasse kann auf diese Eigenschaften zugreifen und sie verändern. Mit dem Schlüsselwort `private` können wir das verhindern.

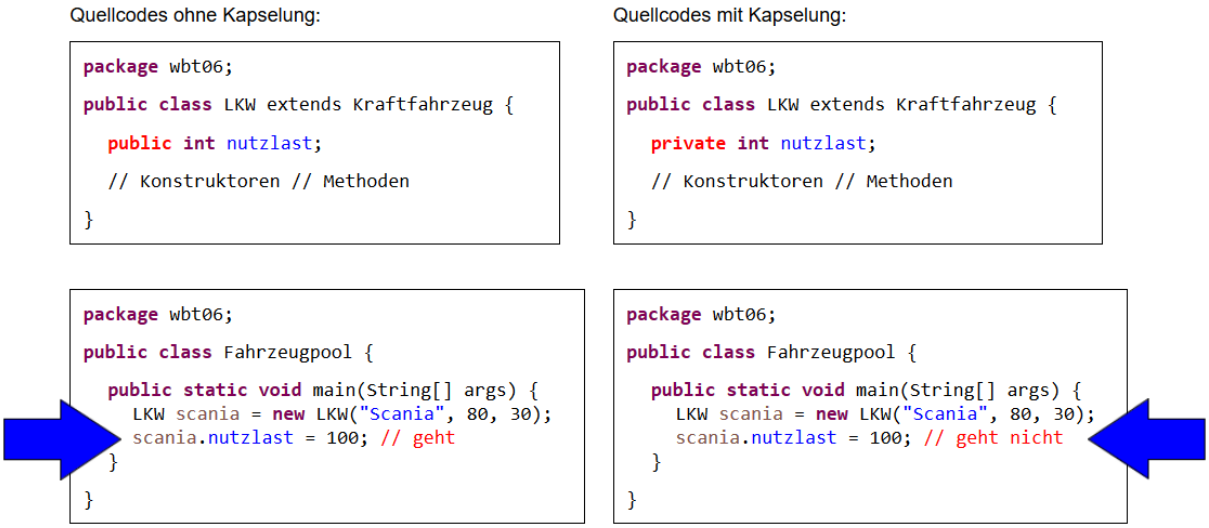


Abb. 50: Manipulation der Eigenschaften mit direktem Zugriff verhindern

6.5 Geheimnisprinzip

Nachdem wir `private` vor die Eigenschaften in den Klassen *Kraftfahrzeug* und *LKW* geschrieben haben, können wir in der Klasse *Fahrzeugpool* auf die Eigenschaften der Objekte nach ihrem Erzeugen nicht mehr direkt zugreifen:

```
package wbt06;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto lamborghini = new Auto("Lamborghini", 400);
		LKW scania = new LKW("Scania", 80, 30);

		lamborghini.maxgeschwindigkeit = 1000000;
		scania.maxgeschwindigkeit = 1000000;
	}
}
```

Bildlich gesprochen, haben wir also die Eigenschaften unserer Objekte in einer Kapsel verborgen. Man spricht daher in der Objektorientierung auch vom sog. Geheimnisprinzip.

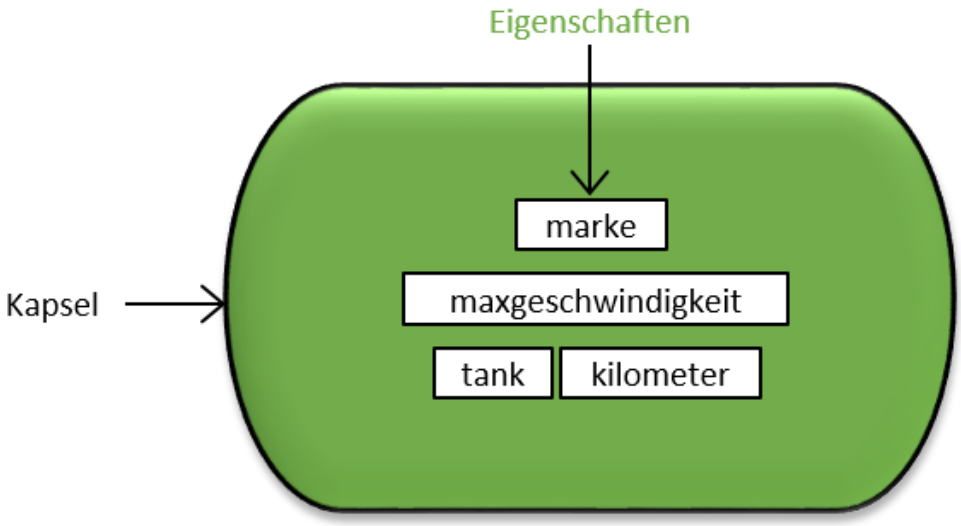


Abb. 51: Eigenschaften in einer Kapsel verbergen

6.6 Zugriffsmethoden

Wir wollen die Eigenschaften von Kraftfahrzeugen und LKW aber nicht für immer in ihrer "Kapsel verschließen". Vielmehr wollen wir den Zugriff auf die einzelnen Eigenschaften kontrollieren. So soll z. B. das Setzen einer Höchstgeschwindigkeit bis 500 km/h erlaubt sein. Dafür schreiben wir sog. Zugriffsmethoden:

```
package wbt06;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto ferrari = new Auto("Ferrari", 350, 0, 0);

		System.out.println(ferrari.getTank());
		ferrari.setTank(50);
		System.out.println(ferrari.getTank());
	}

}
```

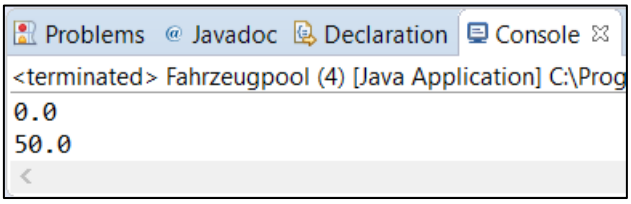


Abb. 52: Getter- und setter-Methoden ausführen

Getter-Methoden haben die Aufgabe, Eigenschaften anzuzeigen. Mit Setter-Methoden können die Eigenschaften verändert werden.

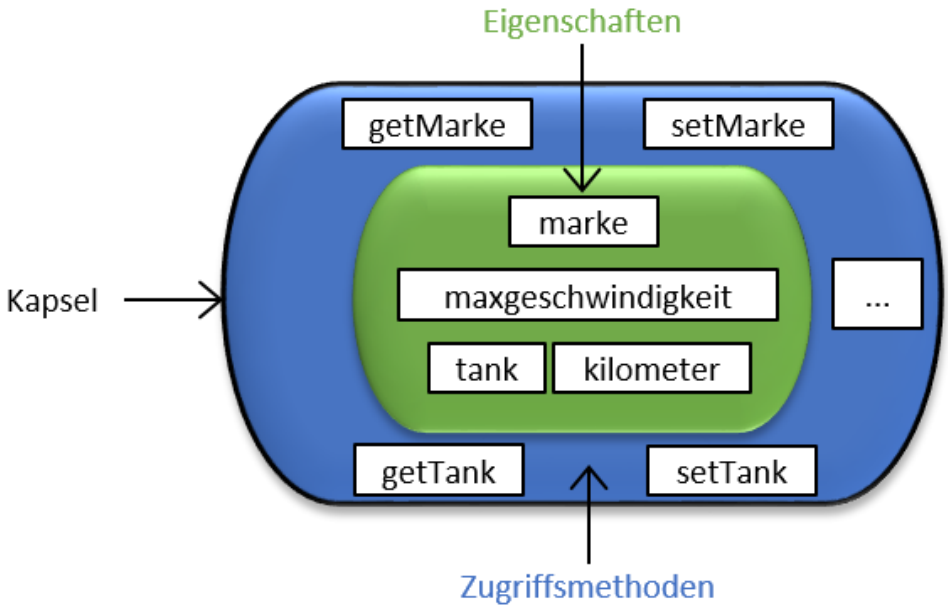


Abb. 53: Mit Zugriffsmethoden den Zugriff auf Eigenschaften kontrollieren

6.7 Die getter-Methode

Die Aufgabe der getter-Methode ist es, gekapselte Eigenschaften eines Objekts anzuzeigen. Der Name leitet sich aus dem englischen Wort "to get" (etwas bekommen, erhalten) ab. Gemäß der Java-Quellcode-Konventionen setzt sich der Name einer getter-Methode immer aus dem Wort "get" und dem Namen der Eigenschaft zusammen. Dabei wird der erste Buchstabe der Eigenschaft großgeschrieben. Getter-Methoden werden immer nach dem gleichen Schema geschrieben:

```
public <Datentyp> get<Eigenschaft>(){

return <eigenschaft>;

}
```

Nach diesem Schema schreiben wir jetzt die getter-Methoden für alle Eigenschaften der Klassen LKW und Kraftfahrzeug:

```
package wbt06;

public class LKW extends Kraftfahrzeug {

	private int nutzlast;

	// Konstruktoren // Methoden

	public int getNutzlast() {
		return nutzlast;
```

```
	}

}
```

```
package wbt06;

public abstract class Kraftfahrzeug {

	private String marke; // Marke
	private int maxgeschwindigkeit; // Hoechstgeschwindigkeit
	private double tank; // Tankstand
	private int kilometer; // Kilometerstand

	// Konstruktoren // Methoden

	public String getMarke() {
		return marke;
	}

	public int getMaxgeschwindigkeit() {
		return maxgeschwindigkeit;
	}

	public double getTank() {
		return tank;
	}

	public int getKilometer() {
		return kilometer;
	}

}
```

6.8 Aufruf der getter-Methode

Nachdem wir die Eigenschaften eines Kraftfahrzeugs gekapselt haben, können die Subklassen *Auto* und *LKW* in ihrer anhalten()-Methode nicht mehr direkt auf den Tank- und Kilometerstand zugreifen. Die anhalten()-Methoden können erst wieder aufgerufen werden, wenn wir den Tank- und Kilometerstand über die getter-Methoden abfragen (`super.get`<Eigenschaft>`()`). Das heißt, wir müssen die beiden anhalten()-Methoden anpassen:

```
package wbt06;

public class Auto extends Kraftfahrzeug {

	@Override
	public void anhalten() {
		System.out.println("Das Auto hat angehalten. Kilometerstand: "
			+ super.getKilometer() + " Kilometer. Tankstand: "
			+ super.getTank() + " Liter.");
	}
```

```
}
```

```
package wbt06;

public class LKW extends Kraftfahrzeug {

	@Override
	public void anhalten() {
		System.out.println("Der LKW hat angehalten. Kilometerstand: "
			+ super.getKilometer() + " Kilometer. Tankstand: "
			+ super.getTank() + " Liter.");
	}

}
```

Nach unserer Anpassung, können wir die anhalten()-Methoden wieder aufrufen. Auch die getter-Methoden können wir testen. Als erstes erzeugen wir in der Klasse *Fahrzeugpool* Objekte aus den Klassen *Auto* und *LKW*. Danach rufen wir die beiden anhalten()-Methoden auf. Zudem zeigen wir die Maximalgeschwindigkeiten mit `public int getMaxgeschwindigkeit()` und die Nutzlast mit `public int getNutzlast()` an:

```
package wbt06;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto lamborghini = new Auto("Lamborghini", 400);
		LKW scania = new LKW("Scania", 80, 30);

		lamborghini.anhalten();
		scania.anhalten();

		System.out.println(lamborghini.getMaxgeschwindigkeit());
		System.out.println(scania.getMaxgeschwindigkeit());
		System.out.println(scania.getNutzlast());
	}

}
```

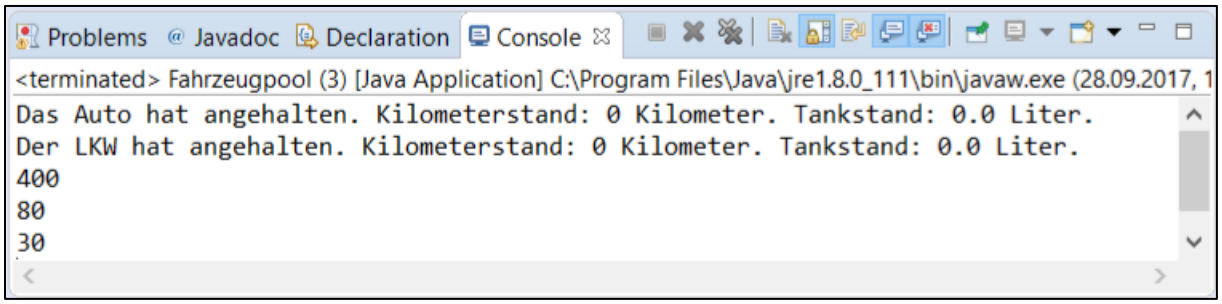


Abb. 54: Methoden „anhalten“, „getMaxgeschwindigkeit“ und „getNutzlast“ ausführen

6.9 Die setter-Methode

Wir wollen die gekapselten Eigenschaften unserer Objekte aber nicht nur anzeigen lassen, sondern auch verändern. Dafür schreiben wir die sog. setter-Methode. Der Name leitet sich aus dem englischen Wort "to set“ (etwas festlegen) ab. Gemäß der Java Quellcode-Konventionen setzt sich der Name einer setter-Methode immer aus dem Wort "set" und dem Namen der Eigenschaft zusammen. Dabei wird der erste Buchstabe der Eigenschaft großgeschrieben.

Einer setter-Methode wird ein Parameter übergeben, der vom gleichen Datentyp ist, wie die Eigenschaft selbst. Der Parameter wird der Eigenschaft der Klasse zugewiesen. Wir schreiben in den Klassen *Kraftfahrzeug* und *LKW* für alle Eigenschaften die entsprechende setter-Methode. Dabei verwenden wir das folgende Schema:

```
public void set<Eigenschaft>(<Datentyp> parameter){

this.<eigenschaft> = parameter;

}
```

```java
package wbt06;

public abstract class Kraftfahrzeug {

	private String marke; // Marke
	private int maxgeschwindigkeit; // Hoechstgeschwindigkeit
	private double tank; // Tankstand
	private int kilometer; // Kilometerstand

	// Konstruktoren // Methoden // getter-Methoden

	public void setMarke(String marke) {
		this.marke = marke;
	}

	public void setMaxgeschwindigkeit(int maxgeschwindigkeit) {
		this.maxgeschwindigkeit = maxgeschwindigkeit;
	}

	public void setTank(double tank) {
		this.tank = tank;
	}

	public void setKilometer(int kilometer) {
		this.kilometer = kilometer;
	}

}
```

```
package wbt06;

public class LKW extends Kraftfahrzeug {

	private int nutzlast;

	// Konstruktoren // Methoden // getter-Methoden

	public void setNutzlast(int nutzlast) {
		this.nutzlast = nutzlast;
	}

}
```

6.10 Aufruf der setter-Methode

Nachdem wir für alle Eigenschaften in den Klassen *Kraftfahrzeug* und *LKW* die setter-Methode geschrieben haben, wollen wir diese nun testen. Dafür erzeugen wir in der Klasse *Fahrzeugpool* Objekte aus den Klassen *Auto* und *LKW*. Anschließend rufen wir mit beiden Objekten die setter-Methode der Höchstgeschwindigkeit auf und übergeben "1000000" als Parameter. Danach überprüfen wir unsere Eingabe mit der toString()-Methode:

```
package wbt06;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto lamborghini = new Auto("Lamborghini", 400);
		LKW scania = new LKW("Scania", 80, 30);

		lamborghini.setMaxgeschwindigkeit(1000000);
		scania.setMaxgeschwindigkeit(1000000);

		System.out.println(lamborghini.toString());
		System.out.println(scania.toString());
	}

}
```

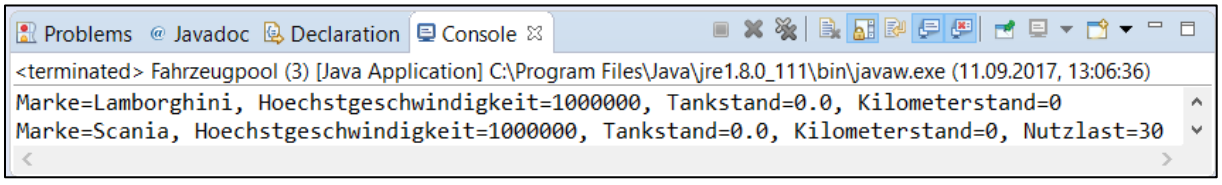


Abb. 55: Die Methoden „setMaxgeschwindigkeit“ und „toString“ ausführen

6.11 Setter-Methode mit Kontrollmechanismus

Wie Sie gerade gesehen haben, können wir unseren Eigenschaften beim Aufruf der setter-Methode noch immer ungewünschte Parameter übergeben. Als nächstes wollen wir in `setMaxgeschwindigkeit()` einen Kontrollmechanismus einbauen, der verhindert, dass Werte größer als 500 km/h zugewiesen werden. Dafür schreiben wir in der setter-Methode eine entsprechende if-Anweisung. Diese überprüft den übergebenen Parameter auf seine Gültigkeit:

```
package wbt06;

public abstract class Kraftfahrzeug {

	private String marke; // Marke
	private int maxgeschwindigkeit; // Hoechstgeschwindigkeit
	private double tank; // Tankstand
	private int kilometer; // Kilometerstand

	// Konstruktoren // Methoden // getter-Methoden // setter-Methoden

	public void setMaxgeschwindigkeit(int maxgeschwindigkeit) {
		if (maxgeschwindigkeit <= 500) {
			this.maxgeschwindigkeit = maxgeschwindigkeit;
		} else
			System.out.println("Die Höchstgeschwindigkeit übersteigt
					500 km/h.");
	}

}
```

Nachdem wir unseren Kontrollmechanismus eingefügt haben, können wir nur noch Höchstgeschwindigkeiten zuweisen, die kleiner als 500 km/h sind. Um dies zu überprüfen, erzeugen wir Objekte aus den Klassen *Auto* und *LKW*. Wenn wir diesen eine Höchstgeschwindigkeit von z. B. 1.000.000 km/h zuweisen, erhalten wir die in der setter-Methode definierte Fehlermeldung und der Wert wird nicht zugewiesen. Wählen wir aber eine Höchstgeschwindigkeit, die kleiner als 500 km/h ist, wird der Wert zugewiesen:

```
public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto lamborghini = new Auto("Lamborghini", 400);
		LKW scania = new LKW("Scania", 80, 30);

		lamborghini.setMaxgeschwindigkeit(1000000);
		System.out.println(lamborghini.toString());
		lamborghini.setMaxgeschwindigkeit(450);
		System.out.println(lamborghini.toString());

		scania.setMaxgeschwindigkeit(1000000);
		System.out.println(scania.toString());
		scania.setMaxgeschwindigkeit(100);
		System.out.println(scania.toString());
	}

}
```

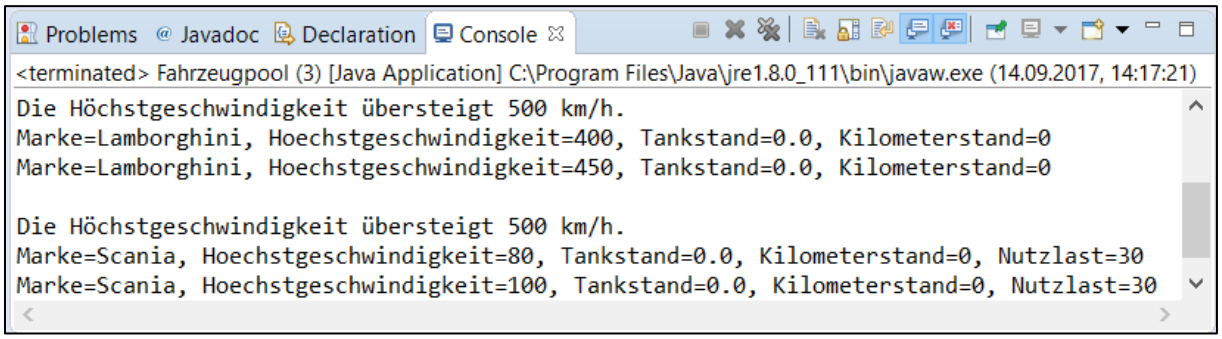


Abb. 56: Methode „setHoechstgeschwindigkeit“ mit Kontrollmechanismus ausführen

6.12 Lessons Learned

Horst Schäfer: „In der heutigen Schulung haben Sie gelernt, dass Eigenschaften gekapselt werden können, um sie vor einem unkontrollierten Zugriff von außen zu schützen. In einem ersten Schritt wurden die Eigenschaften mit dem Schlüsselwort `private` markiert. In einem zweiten Schritt erzeugten wir die sog. getter-und setter-Methoden. Diese kennzeichneten wir als `public`, damit sie von allen anderen Klassen aufgerufen werden können. Mit geeigneten Kontrollmechanismen in den getter- und setter-Methoden, können wir die Eingabe ungewünschter Werte verhindern.“

7. Mit einem Interface Autos und LKW mietbar machen

7.1 Der sechste Schulungstag

Horst Schäfer: „Hallo Frau Schmitt, ich heiße Sie als Praktikantin zum sechsten Tag unserer Schulung "Objektorientierte Programmierung mit Java und Eclipse" herzlich willkommen! In den vergangenen Tagen haben Sie bereits die Konzepte Abstraktion, Überschreibung, Polymorphie und Kapselung am Beispiel einer Autovermietung kennengelernt. Dieses Beispiel werden wir auch in der heutigen Schulung für das Konzept der Interfaces heranziehen.“

7.2 Das Paket “wbt07“ erzeugen

Heute wollen wir an den Klassen *Auto*, *Fahrzeugpool*, *Kraftfahrzeug* und *LKW* weiterarbeiten. Diese haben wir bereits am letzten Schulungstag im Paket "wbt06" erstellt. Bitte erzeugen Sie ein neues Paket "wbt07" in Ihrem Java-Projekt "ObjektorientierteProgrammierung". Kopieren Sie danach die Klassen *Auto*, *Fahrzeugpool*, *Kraftfahrzeug* und *LKW* im Paket "wbt05" und fügen Sie diese in das Paket "wbt07" ein.

7.3 Interfaces

Horst Schäfer: „Bis jetzt haben wir Java-Quellcode immer innerhalb einer Klasse geschrieben. Aus dieser Klasse haben wir Objekte erzeugt und Objektmethoden aufgerufen. Neben Klassen können in Java aber auch sog. Interfaces erzeugt werden. Interfaces enthalten ausschließlich Konstanten und abstrakte Methoden. Aus einem Interface können keine Objekte erzeugt werden. Dabei hat ein Interface auch keinen Konstruktor, weil Konstruktoren nur in (abstrakten) Klassen stehen. Ihre Aufgabe ist es, anderen Klassen bestimmte Verhaltensweisen vorzuschreiben. Sobald eine Klasse ein Interface verwendet, geht sie damit einen Vertrag ein. Sie muss alle im Interface enthaltenen abstrakten Methoden und Konstanten berücksichtigen.“

7.4 Interfaces in Java

Ein Interface wird genau wie eine Klasse mit einem Schlüsselwort definiert. Das Schlüsselwort lautet `interface`: `public interface` <InterfaceName> `{...}`

Ein Interface enthält immer abstrakte Methoden und/oder Konstanten. Die Aufgabe des Interfaces ist es, anderen Klassen vorzuschreiben, wie sie sich verhalten sollen. Um einer Klasse ein Interface vorzuschreiben, wird das Schlüsselwort `implements` verwendet:

`public` <KlassenName> `implements` <InterfaceName> `{...}`

```java
package wbt07;

public interface Interface {

	public abstract void halloWelt();

}
```

```java
package wbt07;

public class Klasse {

	// Eigenschaften // Konstruktor // Methoden

	public void halloWelt() {
		System.out.println("Hallo, Welt!");
	}

}
```

7.5 Das Interface Mietbar definieren

Um den Nutzen von Interfaces zu verdeutlichen, wollen wir jetzt ein Interface schreiben. Das Interface soll die Kraftfahrzeuge in unserer Fahrzeugvermietung mietbar machen. Daher nennen wir das Interface "Mietbar". Ein Interface ist ein Vertrag, der anderen Klassen vorschreibt, wie sie sich verhalten sollen. Also müssen wir uns im ersten Schritt überlegen, welche Verhaltensweisen das Interface Mietbar vorscheiben soll:

- Fahrzeuge mieten → `public abstract void mieten()`
- Fahrzeuge zurückgeben → `public abstract void rueckgabe()`

7.6 Interfaces im UML-Klassendiagramm

Im UML-Klassendiagramm schreibt man die Bezeichnung <<interface>> hinter den Namen eines Interface. Um zu kennzeichnen, dass eine Klasse ein Interface implementiert, wird ein gestrichelter Pfeil von der Klasse zum Interface gezogen.

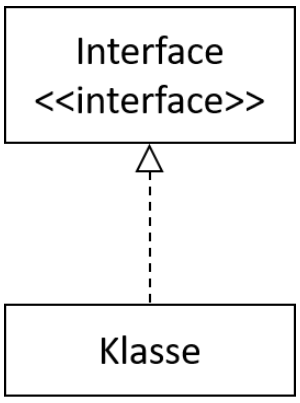


Abb. 57: Interfaces im UML-Klassendiagramm

Alle Methoden in einem Interface sind (implizit) `abstract` und `public`. In unserem Beispiel implementiert die Klasse *Kraftfahrzeug* das Interface *Mietbar*.

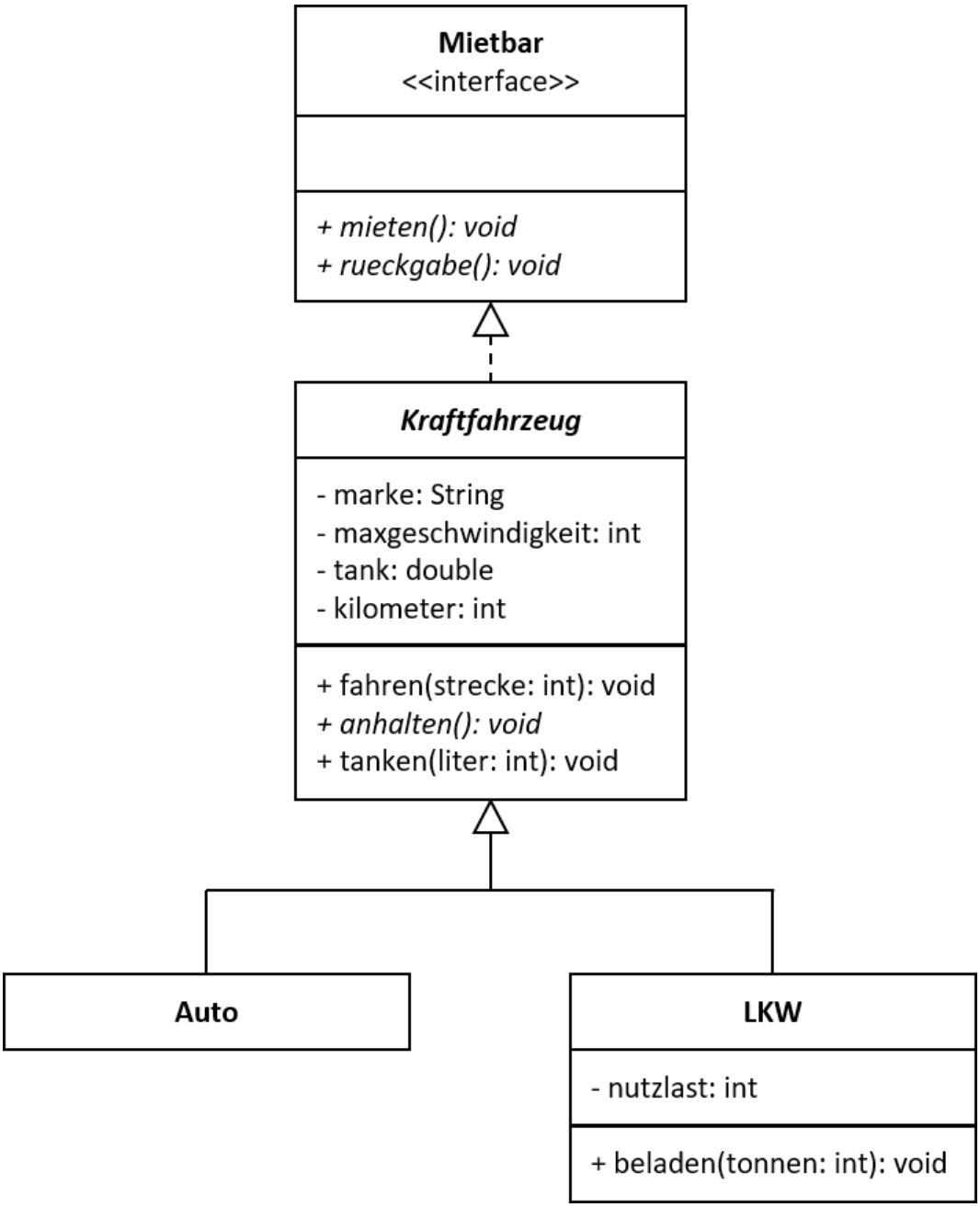


Abb. 58: Das Interface „Mietbar“ im UML-Klassendiagramm

7.7 Das Interface Mietbar schreiben

Jetzt wollen wir das Interface *Mietbar* schreiben. Bitte führen Sie dafür die folgenden beiden Schritte durch:

- Erzeugen Sie zuerst über File/New/Interface im Paket "wbt07" ein Interface mit dem Namen "Mietbar".

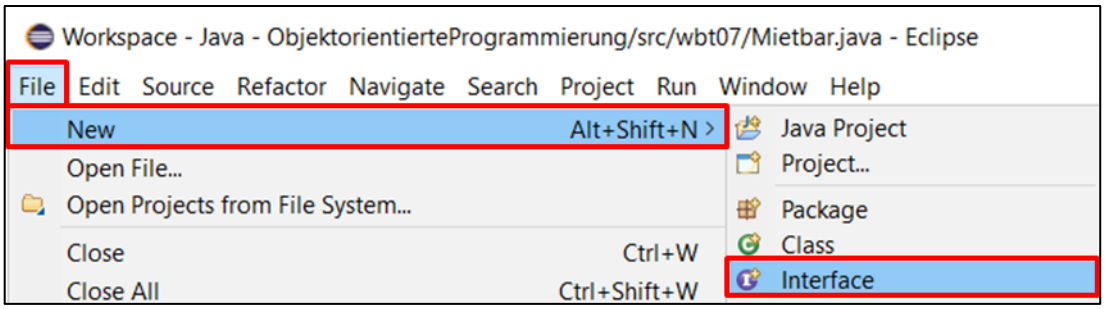


Abb. 59: Das Interface „Mietbar“ mit Eclipse erzeugen

- Übertragen Sie danach die Methoden aus dem UML-Klassendiagramm in das Interface:

```
package wbt07;

public interface Mietbar {

	public abstract void mieten();

	public abstract void rueckgabe();

}
```

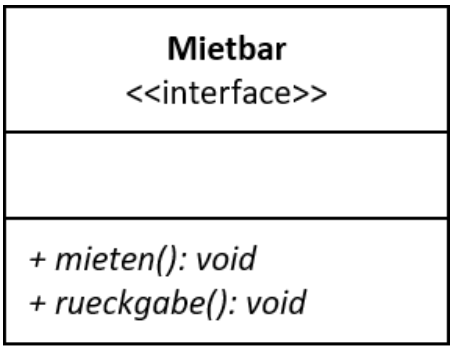


Abb. 60: UML-Klassendiagramm des Interfaces „Mietbar“

7.8 Das Interface Mietbar implementieren

Nachdem wir das Interface *Mietbar* geschrieben haben, soll es die Klasse *Kraftfahrzeug* verwenden. Dafür nutzen wir das Schlüsselwort `implements`:

```
package wbt07;

public abstract class Kraftfahrzeug implements Mietbar {

	// Eigenschaften // Konstruktor // Methoden

	@Override
	public abstract void mieten();

	@Override
	public abstract void rueckgabe();

}
```

Ab jetzt sind alle Kraftfahrzeuge dazu verpflichtet, Methoden anzubieten, die sie mietbar machen oder ihre Rückgabe ermöglichen. Da Autos und LKW Kraftfahrzeuge sind, müssen sie den Vertrag des Interfaces Mietbar erfüllen. Sie müssen die abstrakten Methoden implementieren. Also einen Methodenblock definieren:

```
package wbt07;

public class Auto extends Kraftfahrzeug {

	// Konstruktoren // Methoden

	@Override
	public void mieten() {
		System.out.println("Sie haben das Auto gemietet.");
	}

	@Override
	public void rueckgabe() {
		System.out.println("Sie haben das Auto zurückgegeben.");
	}

}
```

```
package wbt07;

public class LKW extends Kraftfahrzeug {

	// Eigenschaft // Konstruktoren // Methoden

	@Override
	public void mieten() {
		System.out.println("Sie haben den LKW gemietet.");
	}

	@Override
	public void rueckgabe() {
		System.out.println("Sie haben den LKW zurückgegeben.");
	}

}
```

7.9 Autos und LKW mieten

Nachdem wir das Interface *Mietbar* implementiert haben, wollen wir es testen. Dafür erzeugen wir in der Klasse *Fahrzeugpool* Objekte aus den Klassen *Auto* und *LKW*. Danach mieten wir das Auto und den LKW und geben diese wieder zurück:

```
package wbt07;

public class Fahrzeugpool {

	public static void main(String[] args) {
		Auto lamborghini = new Auto("Lamborghini", 400);
		LKW scania = new LKW("Scania", 80, 30);

		lamborghini.mieten();
		scania.mieten();

		lamborghini.rueckgabe();
		scania.rueckgabe();
	}

}
```

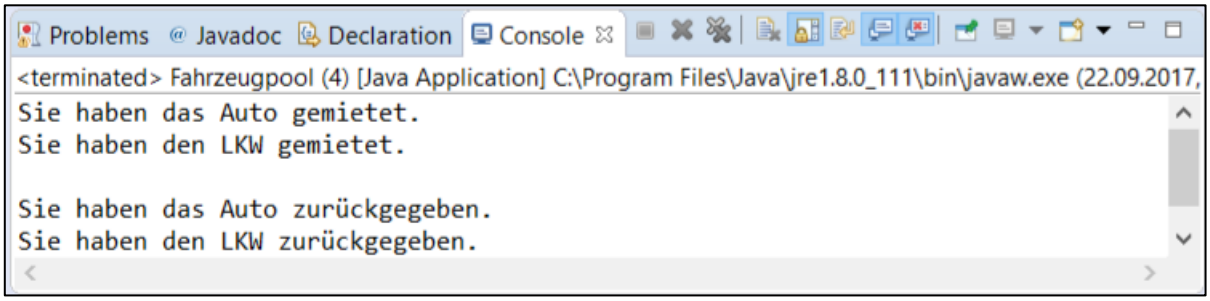


Abb. 61: Die Methoden „mieten“ und „rueckgabe“ ausführen

7.10 Mehrfachvererbung mit Interfaces

Anna Schmitt: „Ich frage mich, warum wir das Interface *Mietbar* schreiben. Wir könnten doch einfach eine neue Klasse *Mietbar* erzeugen. Die Klassen *Auto* und *LKW* könnten dann von dieser Klasse erben.“

Horst Schäfer: „Das ist in Java leider nicht möglich, Frau Schmitt. In Java kann eine Klasse immer nur von 1 weiteren Klasse erben. Eine Klasse kann jedoch mehrere Interfaces implementeren. Interfaces bieten uns also die Möglichkeit, eine Art der "Mehrfachvererbung" in Java zu realisieren.“

7.11 Klassen versus Interfaces

Wie Sie heute gelernt haben, gibt es in Java nicht nur Klassen, sondern auch Interfaces. Abschließend wollen wir Interfaces und Klassen anhand verschiedener Merkmale gegenüberstellen.

Merkmal	Klasse	Interface
Definition	class	interface
Verwendung	extends	implements
Inhalte	Konstanten, nicht-abstrakte Methoden, Eigenschaften, statische Variablen, Konstruktoren, statische Methoden	Konstanten, abstrakte Methoden
Anwendungsbereich	Bauplan für Objekte	Verträge, die Verhaltensweisen vorgeben
Besonderheit	eine direkte Superklasse	„Mehrfachvererbung“ realisierbar

Tab. 2: Klassen versus Interfaces

7.12 Lessons learned

In der heutigen Schulung haben Sie gelernt, dass es in Java neben Klassen auch Interfaces gibt. Interfaces weisen im Vergleich zu Klassen die folgenden Unterschiede auf:

- enthalten nur Konstanten (`final`),
- alle Methoden sind `public` und `abstract`,
- haben keine Methodenblöcke (Implementationen) und
- haben keine Konstruktoren.

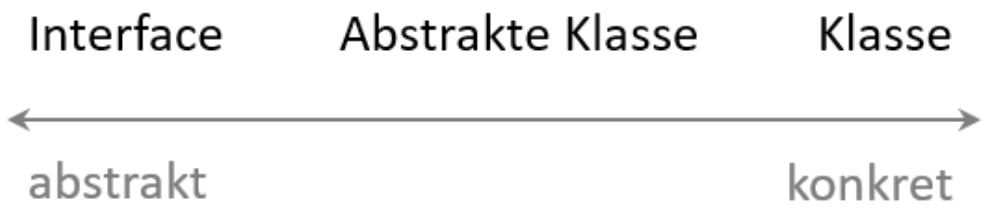


Abb. 62: Einordnung Interface, abstrakte Klasse und Klasse

8. Zusammenfassung am Beispiel des Webshops der Lemonline AG

8.1 Einführung

8.1.1 Der siebte Schulungstag

Horst Schäfer: „Hallo Frau Schmitt, ich heiße Sie als Praktikantin zum siebten Tag unserer Schulung "Objektorientierte Programmierung mit Java und Eclipse" herzlich Willkommen! In den vergangenen Tagen haben Sie bereits viel über das Thema "Objektorientierung" gelernt. In der heutigen Schulung werden wir dieses Wissen zusammenfassen. Danach wollen wir die Konzepte der Objektorientierung nutzen, um prototypisch die Elemente eines Webshops für die Lemonline AG in Java zu programmieren.“

8.1.2 Zusammenfassung: Objektorientierung

Horst Schäfer: „Zuerst wollen wir die Inhalte der Schulung "Objektorientierte Programmierung mit Java und Eclipse" zusammenfassen. Sie haben gelernt, dass wir mit Hilfe der Objektorientierung reale Gegenstände als sog. Objekte in einem Programm abbilden können. Jedes Objekt besteht aus Eigenschaften und Verhaltensweisen. Diese müssen wir als erstes in Java-Klasse planen. Dabei nutzen wir zur Visualisierung eine Modellierungssprache, wie z. B. UML. Mithilfe der UML modellieren wir die Eigenschaften und Verhaltensweisen eines Objekts in einem UML-Klassendiagramm. Danach programmieren wir den Java-Quellcode in einer Java-Klasse.“

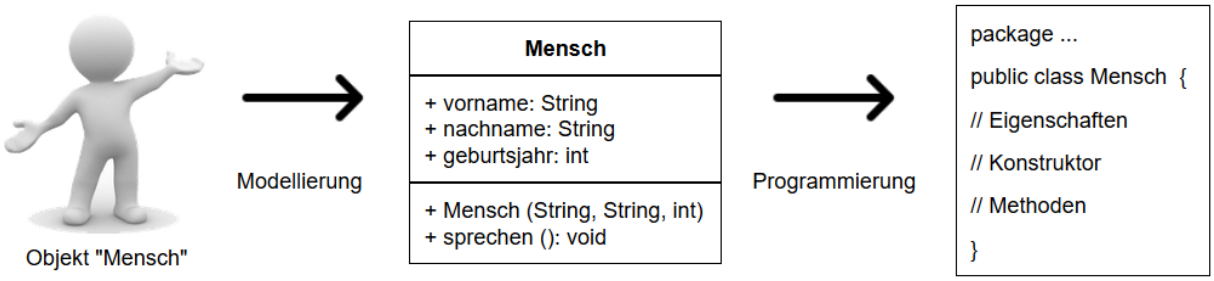


Abb. 63: Zusammenfassung: objektorientierte Programmierung

8.1.3 Zusammenfassung: Konzepte der Objektorientierung

1. **Vererbung**: Gemeinsame Eigenschaften und Verhaltensweisen ähnlicher Klassen werden in einer Superklasse zusammengefasst und an die Subklassen vererbt.
2. **Abstraktion:** Abstrakte Klassen werden verwendet, wenn aus einer Klasse keine Objekte erzeugt werden sollen, sie aber trotzdem ihre Eigenschaften und Verhaltensweisen vererben soll.

3. **Überschreibung**: Durch Überschreibung wird eine Methode in der Subklasse ersetzt, die bereits in einer übergeordneten Klasse deklariert bzw. definiert wurde.
4. **Polymorphie**: Eine Methode kann in vielen Gestalten in einer oder verschiedenen Klassen auftreten.
5. **Kapselung**: Durch die Kapselung von Eigenschaften kann nur noch die Klasse, in der die Eigenschaft steht, direkt auf diese zugreifen.
6. **Interfaces**: Interfaces sind keine Klassen sondern "Verträge", die ausschließlich abstrakte Methoden und Konstanten beinhalten.

8.2 Den Webshop planen

8.2.1 Präsentation der Lemonline AG

Horst Schäfer: „Heute wollen wir mit den Konzepten der Objektorientierung prototypisch einen Webshop für die Lemonline AG in Java programmieren. Wie Sie bereits gelernt haben, müssen wir dazu im ersten Schritt unsere Objekte planen. Für unseren Bauplan müssen wir die Anforderungen an unseren Webshop analysieren. Dafür sehen wir uns jetzt eine Präsentation der Lemonline AG an, um weitere Informationen über das Unternehmen und sein Produktportfolio zu erhalten."

8.2.2 Die Lemonline AG

Die Lemonline AG ist einer der führenden Hersteller von Smartphones und Tablet-PCs in Deutschland. Der Hersteller beliefert sowohl Endkunden als auch Unternehmen (B2C, B2B). Der Unternehmensname leitet sich aus den englischen Wörtern „Lemon" (Zitrone) und „online" (verbunden, betriebsbereit, aktiv) ab und deutet auf die Unternehmensziele der Lemonline AG – frische Produktideen und ständige Erreichbarkeit des Kundenservices. Hier finden Sie weitere Informationen über die historische Entwicklung der Lemonline AG:

- **2003**: Nach der Gründung im Jahr 2003 in Frankfurt am Main wurde die Lemonline AG zunächst als Anbieter von qualitativ hochwertigen Mobiltelefonen bekannt.
- **2008**: Im Jahr 2008 fasste das Unternehmen auf dem stark wachsenden Smartphone-Markt Fuß und erzielte einen erstaunlichen Erfolg.
- **2012**: Heute produziert die Lemonline AG neben Mobiltelefonen und Smartphones auch Tablet-PCs - elektronische Geräte, die auf unterschiedliche Weise die Eigenschaften und Funktionen eines Mobiltelefons und eines tragbaren Computers miteinander verbinden.

Die Lemonline AG beschäftigt mehr als 1000 Mitarbeiter an 3 Standorten in Deutschland:

- Hamburg
  - Aufbau: Jahr 2006
  - Mitarbeiter 2016: 553
  - Organisatorischer Aufbau: F&E, Produktion, Lager
- Frankfurt am Main
  - Gründung: Jahr 2003
  - Mitarbeiter 2016: 450
  - Organisatorischer Aufbau: Management, Marketing, Vertrieb, Service
- Stuttgart
  - Aufbau: Jahr 2010
  - Mitarbeiter 2016: 56
  - Lemon-Store: Smartphones, Tablet-PCs, Mobiltelefone, Zubehör

8.2.3 Produkte der Lemonline AG

Die Lemonline AG produziert in ihrer Niederlassung in Hamburg sowohl klassische Mobiltelefone, als auch leistungsstarke Smartphones und multifunktionelle Tablet-PCs. Somit sind die Wünsche aller Zielgruppen des Unternehmens bestens abgedeckt.

- Mobiltelefone
  - Features
    - Zuverlässige Kommunikation
    - Eingeschränkte Funktionen
    - Einfache Bedienung
  - Modelle
    - Lemon Classic
    - Lemon Star
    - Lemon Light
- Smartphones
  - Features
    - Moderne Kommunikation
    - Mobiles Entertainment
    - Großer Funktionsumfang

- Modelle
  - Lemon Five
  - Lemon Wave
  - Lemon Sharp

- Tablet-PCs
  - Features
    - Moderne Kommunikation
    - Mobilität und Flexibilität
    - Großer Funktionsumfang
  - Modelle
    - Lemon Top
    - Lemon Optimum

8.2.4 Anwendungsfälle des Webshops

Wie Sie gerade gelernt haben, hat die Lemonline AG sowohl B2B als auch B2C-Kunden. Sie verkauft Mobiltelefone, Smartphones und Tablets. Für unser Programm bedeutet das, dass wir Kunden als Objekte erzeugen müssen. Diese können entweder ein Privat- oder ein Geschäftskunde sein. In unserem Webshop sollen sich Kunden ihre Kundendetails anzeigen lassen können. Zudem sollen sie die Möglichkeit haben, Artikel wie Mobiltelefone, Smartphones und Tablets im Webshop anzusehen sowie zu bestellen.

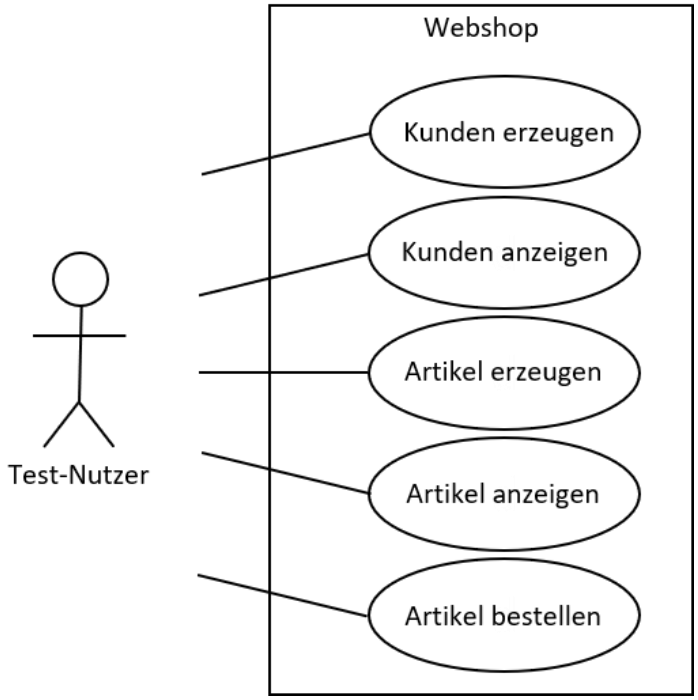


Abb. 64: Anwendungsfälle des Webshops

8.2.5 Spezifikation der Anwendungsfälle

Nachdem wir die Anwendungsfälle unseres Webshops definiert haben, wollen wir diese spezifizieren. Wir wollen herausfinden:

- welche Klassen mit welchen Eigenschaften und Verhaltensweisen erforderlich sind,
- ob Vererbungsbeziehungen vorliegen,
- eine Superklasse ggfs. eine abstrakte Klasse ist,
- welche Beziehungen zwischen den Klassen bestehen.

Anwendungsfälle:

- **Kunden erzeugen**: Die Lemonline AG hat sowohl B2B- als auch B2C-Kunden. Daher benötigen wir die Klassen *Kunde*, *Privatkunde* und *Geschaeftskunde*. Da unsere Kunden immer entweder Privat- oder Geschäftskunden sind, ist die Klasse *Kunde* abstrakt. Alle Kunden haben einen Namen, eine Kundennummer und eine Adresse. Privatkunden haben zusätzlich einen Vornamen und Geschäftskunden eine UmsatzsteuerID, kurz UstID.
- **Kunden anzeigen**: Kundendetails sollen mit der toString()-Methode angezeigt werden. Dabei soll allen Kunden ihr Name, ihre Kundennummer und ihre Adresse angezeigt werden. Zusätzlich soll der Vorname für Privatkunden und für Geschäftskunden die UstID ausgegeben werden.
- **Artikel erzeugen:** Die Lemonline AG produziert Mobiltelefone, Smartphones und Tablets. Daher benötigen wir die Klasse *Artikel*. Alle Artikel haben eine Artikelnummer, eine Artikelbezeichnung, eine Artikelgruppe (Mobiltelefon, Smartphone, Tablet) und einen Nettopreis. Die Klasse *Artikel* geht einen Vertrag mit dem Interface *Kaeuflich* ein. Dieses stellt den regulären Steuersatz (19 %) und die Verhaltensweise “Bruttopreis berechnen“ zur Verfügung. Damit wir in unserem Webshop gerundete Preise angezeigt bekommen, wollen wir die Methode "round" der Klasse *Math* nutzen. Dafür muss die Klasse *Artikel* die Klasse *Math* importieren.
- **Artikel anzeigen:** Artikeldetails sollen mit der toString()-Methode angezeigt werden. Dabei soll immer die Artikelnummer, die Artikelbezeichnung, die Artikelgruppe (Mobiltelefon, Smartphone, Tablet) und der Nettopreis angezeigt werden.
- **Artikel bestellen:** Kunden sollen im Webshop die Möglichkeit haben, einen bzw. zwei Artikel zu bestellen. Dabei soll bei jeder Bestellung immer der (die) Artikel mit der (den) gewünschten Anzahl(en) angegeben werden. Die Klasse *Kunde* soll ihren Subklassen die Verhaltensweisen "bestellen" vorschreiben. Daher sind die bestellen()-Methoden abstrakte Methoden. Nach der Bestellung soll Privatkunden der folgende Satz angezeigt werden:

"Privatkunde <name> hat <anzahl> <artikelbezeichnung> zum Bruttostückpreis von <berechneBruttopreis()> € bestellt."
Geschäftskunden hingegen soll dieser Satz angezeigt werden:
"Geschaeftskunde <Name> hat <anzahl> + <artikelbezeichnung> zum Nettostückpreis von <nettopreis> € bestellt."

8.2.6 UML-Klassendiagramme

Nachdem wir spezifiert haben, welche Eigenschaften und Verhaltensweisen unsere Klassen haben, können wir die UML-Klassendiagramme für die Klassen *Kunde*, *Privatkunde*, *Geschaeftskunde*, *Artikel* und *Kaeuflich* modellieren.

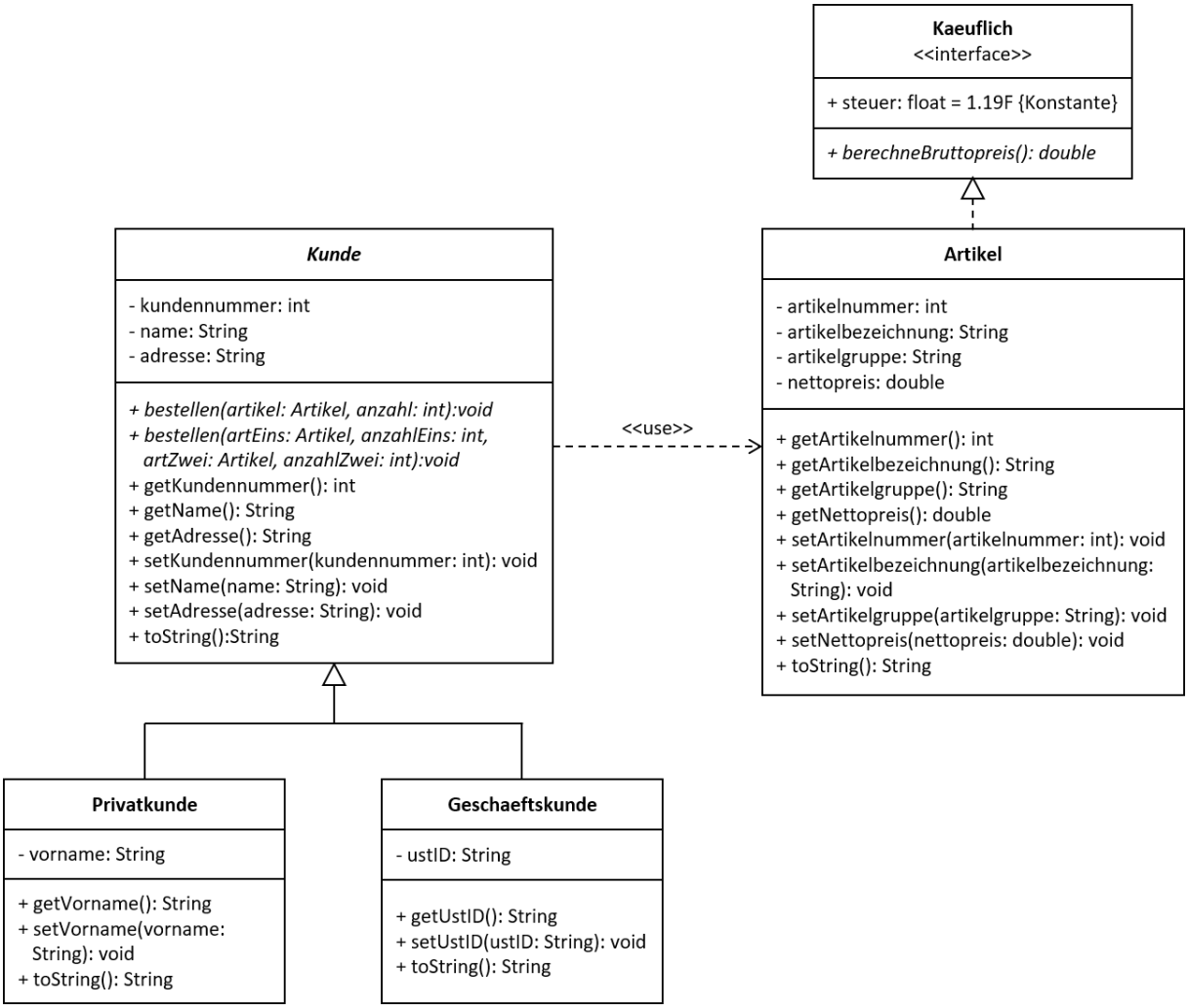


Abb. 65: UML-Klassendiagramme für den Webshop mit Eigenschaften und Methoden

8.3 Java-Quellcode schreiben

8.3.1 Das Paket “wbt08“ erzeugen

Nachdem wir die UML-Klassendiagramme modelliert haben, kann es mit der Programmierung des Java-Quellcodes losgehen.

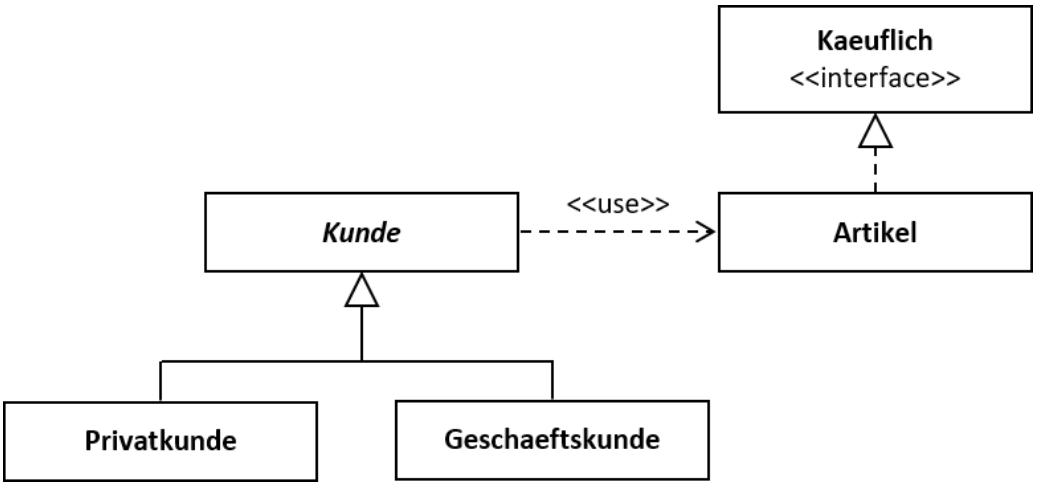


Abb. 66: UML-Klassendiagramme für den Webshop ohne Eigenschaften und Methoden

Bitte erzeugen Sie ein neues Paket "wbt08" mit den im UML-Klassendiagramm angegebenen Klassen. Zusätzlich benötigen wir die Klasse *Webshop* mit einer main-Methode.

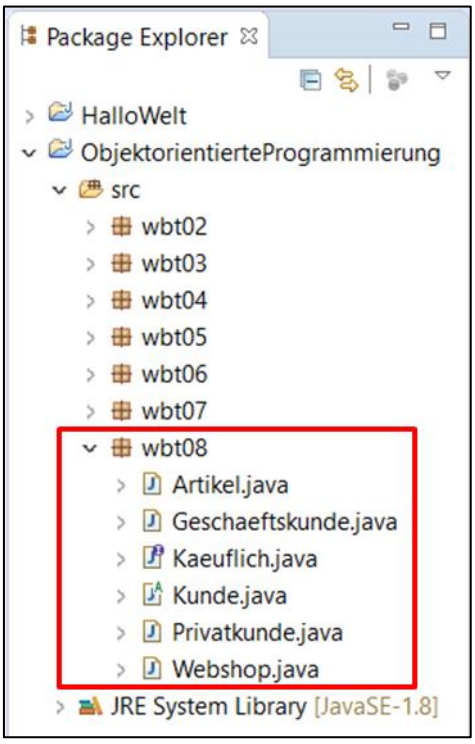


Abb. 67: Das Paket “wbt05“

8.3.2 Die Klasse Kunde

Als erstes schreiben wir den Java-Quellcode für die abstrakte Klasse *Kunde*.

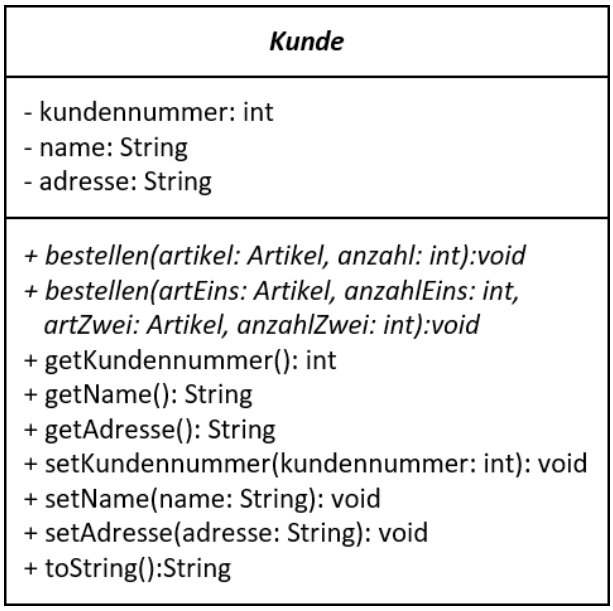


Abb. 68: UML-Klassendiagramm der Klasse „Kunde“

Zuerst übertragen wir die Eigenschaften und Methoden aus dem UML-Klassendiagramm:

```
package wbt08;

public abstract class Kunde {

	private String name;
	private int kundennummer;
	private String adresse;

	public Kunde(String name, int kundennummer, String adresse) {
		this.name = name;
		this.kundennummer = kundennummer;
		this.adresse = adresse;
	}

	public abstract void bestellen(Artikel artikel, int anzahl);

	// ueberladene Methode
	public abstract void bestellen(Artikel artEins, int anzahlEins,
			Artikel artZwei, int anzahlZwei);

	@Override
	public String toString() {
		return "Name=" + this.name + ", Kundennummer="
				+ this.kundennummer + ", Adresse=" + this.adresse;
	}

}
```

Danach schreiben wir die getter- und setter-Methoden für die gekapselten Eigenschaften:

```
package wbt08;

public abstract class Kunde {
```

```
	public int getKundennummer() {
		return kundennummer;
	}

	public String getName() {
		return name;
	}

	public String getAdresse() {
		return adresse;
	}

	public void setKundennummer(int kundennummer) {
		this.kundennummer = kundennummer;
	}

	public void setName(String name) {
		this.name = name;
	}

	public void setAdresse(String adresse) {
		this.adresse = adresse;
	}

}
```

8.3.3 Die Klasse Privatkunde

Als nächstes schreiben wir den Quellcode für die Klasse *Privatkunde*.

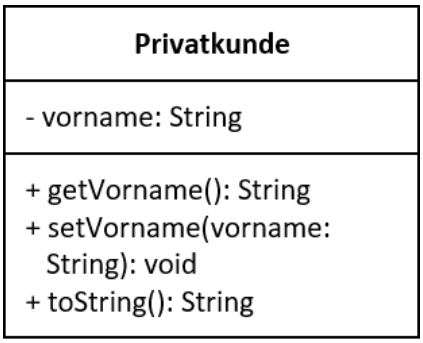


Abb. 69: UML-Klassendiagramm der Klasse „Privatkunde“

Zuerst schreiben wir die Eigenschaften und die getter- und setter-Methoden für die gekapselten Eigenschaften:

```
package wbt08;

public class Privatkunde extends Kunde {

	private String vorname;

	public Privatkunde(String name, int kundenNummer, String adresse,
			String vorname) {
```

```
            super(name, kundenNummer, adresse);
            this.vorname = vorname;
      }

      public String getVorname() {
            return vorname;
      }

      public void setVorname(String vorname) {
            this.vorname = vorname;
      }

}
```

Danach überschreiben wir die von der Superklasse *Kunde* geerbten abstrakten bestellen()-Methoden und die Methode `toString` der Klasse *Object*:

```
package wbt08;

public class Privatkunde extends Kunde {

      @Override
      public void bestellen(Artikel artEins, int anzahlEins) {
            System.out.println("Privatkunde " + super.getName() + " hat "
                  + anzahlEins + " " + artEins.getArtikelbezeichnung()
                  + " zum Bruttostückpreis von "
                  + artEins.berechneBruttopreis() + " € bestellt.");
      }

      @Override
      public void bestellen(Artikel artEins, int anzahlEins,
                  Artikel artZwei, int anzahlZwei) {
            bestellen(artEins, anzahlEins);
            bestellen(artZwei, anzahlZwei);
      }


      @Override
      public String toString() {
            return "Privatkunde: Vorname= " + this.vorname + ", "
                  + super.toString();
      }

}
```

8.3.4 Die Klasse Geschaeftskunde

Analog zur Klasse *Privatkunde* erzeugen wir die Klasse *Geschaeftskunde*.

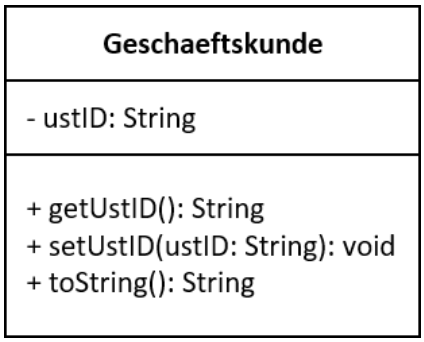


Abb. 70: UML-Klassendiagramm der Klasse „Geschaeftskunde“

Zuerst schreiben wir die Eigenschaften und die getter- und setter-Methoden. Danach überschreiben wir die abstrakten Methoden `bestellen` und die toString()-Methode:

```
public class Geschaeftskunde extends Kunde {

	private String ustID;

	public Geschaeftskunde(String name, int kundenNummer, String adresse,
				String ustID) {
		super(name, kundenNummer, adresse);
		this.ustID = ustID;
	}

	public String getUstID() {
		return ustID;
	}

	public void setUstID(String ustID) {
		this.ustID = ustID;
	}

	@Override
	public void bestellen(Artikel artikel, int anzahl) {
		System.out.println("Geschaeftskunde " + super.getName()
		    + " hat " + anzahl + " " + artikel.getArtikelbezeichnung
		    + " zum Nettostückpreis von " + artikel.getNettopreis()
		    + " € bestellt.");
	}

	@Override
	public void bestellen(Artikel artEins, int anzahlEins,
				Artikel artZwei, int anzahlZwei) {
		bestellen(artEins, anzahlEins);
		bestellen(artZwei, anzahlZwei);
	}

	@Override
	public String toString() {
		return "Geschaeftskunde: " + super.toString() + ", UstID="
			+ this.ustID;
	}
}
```

8.3.5 Das Interface Kaeuflich

Jetzt übertragen wir die Konstante und die abstrakte Methode aus dem UML-Klassendiagramm in das Interface *Kaeuflich*:

```
package wbt08;

public interface Kaeuflich {

	final float steuer = 1.19F;

	public abstract double berechneBruttopreis();

}
```

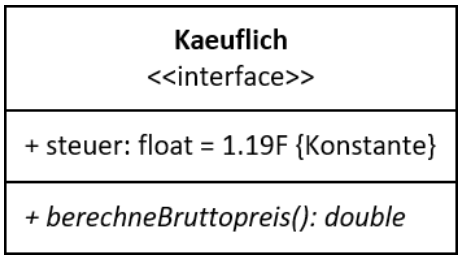


Abb. 71: UML-Klassendiagramm des Interfaces „Kaeuflich“

Die Klasse *Artikel* soll das Interface *Kaeuflich* verwenden und die Verhaltensweise "berechneBruttopreis" implementieren. Wir wollen, dass unsere Bruttopreise im Webshop mit zwei Nachkommastellen angezeigt werden. Dafür müssen wir die Klasse *Math* importieren. Werte vom Datentyp `double` werden mit dem Befehl `Math.round(bruttopreis * 100) / 100.00` gerundet:

```
package wbt08;

import java.lang.Math;

public class Artikel implements Kaeuflich {

	@Override
	public double berechneBruttopreis() {
		return Math.round(nettopreis * steuer * 100) / 100.00;
	}

}
```

8.3.6 Die Klasse Artikel

Nachdem wir `berechneBruttopreis()` implementiert haben, übertragen wir die Eigenschaften aus dem UML-Klassendiagramm und überschreiben die toString()-Methode.

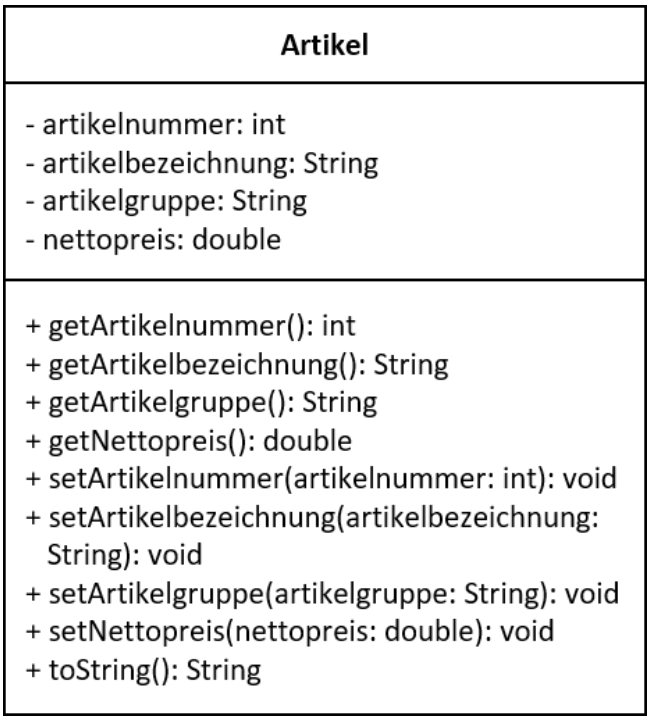


Abb. 72: UML-Klassendiagramm der Klasse „Artikel“

```
package wbt08;

import java.lang.Math;

public class Artikel implements Kaeuflich {

	private int artikelnummer;
	private String artikelbezeichnung;
	private String artikelgruppe;
	private double nettopreis;

	public Artikel(int artikelnummer, String artikelbezeichnung,
			String artikelgruppe, double nettopreis) {
		this.artikelnummer = artikelnummer;
		this.artikelbezeichnung = artikelbezeichnung;
		this.artikelgruppe = artikelgruppe;
		this.nettopreis = nettopreis;
	}


	@Override
	public String toString() {
		return "Artikel: Artikelnummer=" + this.artikelnummer
			+ ", Artikelbezeichnung=" + this.artikelbezeichnung
			+ ", Artikelgruppe=" + this.artikelgruppe
			+ ", Nettopreis=" + this.nettopreis;
	}

}
```

Danach schreiben wir die getter- und setter-Methoden:

```
package wbt08;

import java.lang.Math;

public class Artikel implements Kaeuflich {

	public int getArtikelnummer() {
		return artikelnummer;
	}

	public String getArtikelbezeichnung() {
		return artikelbezeichnung;
	}

	public String getArtikelgruppe() {
		return artikelgruppe;
	}

	public double getNettopreis() {
		return nettopreis;
	}

	public void setArtikelnummer(int artikelnummer) {
		this.artikelnummer = artikelnummer;
	}

	public void setArtikelbezeichnung(String artikelbezeichnung) {
		this.artikelbezeichnung = artikelbezeichnung;
	}

	public void setNettopreis(double nettopreis) {
		this.nettopreis = nettopreis;
	}

	public void setArtikelgruppe(String artikelgruppe) {
		this.artikelgruppe = artikelgruppe;
	}


}
```

8.4 Den Protoyp erzeugen

8.4.1 Webshop vorbereiten

Horst Schäfer: „Nachdem wir den Java-Quellcode für alle Klassen im Paket "wbt08" geschrieben haben, können wir unseren Webshop mit Daten füllen. Danach kann dann die erste Bestellung aufgegeben werden. Zuerst müssen wir jedoch Artikel erzeugen und Kunden erzeugen.“

8.4.2 Artikel erzeugen

Als erstes wollen wir in unserem Webshop Artikel anlegen. Bitte erzeugen Sie in der Klasse Webshop die folgenden Artikel der Lemonline AG mit den angegebenen Artikelnummern und Preisen.

- Mobiltelefone
  - LemonClassic
    - Artikelnummer: 101
    - Nettopreis: 69,99 €
  - Lemon Star
    - Artikelnummer: 102
    - Nettopreis: 89,99 €
  - Lemon Light
    - Artikelnummer: 103
    - Nettopreis: 109,99 €
- Smartphones
  - Lemon Five
    - Artikelnummer: 201
    - Nettopreis: 149,99 €
  - Lemon Wave
    - Artikelnummer: 202
    - Nettopreis: 199,99 €
  - Lemon Sharp
    - Artikelnummer: 203
    - Nettopreis: 249,99 €
- Tablet-PCs
  - Lemon Top
    - Artikelnummer: 301
    - Nettopreis: 499,99 €
  - Lemon Optimum
    - Artikelnummer: 302
    - Nettopreis: 999,99 €

```
package wbt08;

public class Webshop {

	public static void main(String[] args) {
		Artikel lemonClassic = new Artikel(101, "Lemon Classic",
			"Mobiltelefon", 69.99);
		Artikel lemonStar = new Artikel(102, "Lemon Star",
			"Mobiltelefon", 89.99);
		Artikel lemonLight = new Artikel(103, "Lemon Light",
			"Mobiltelefon", 109.99);
		Artikel lemonFive = new Artikel(201, "Lemon Five",
			"Smartphone", 149.99);
		Artikel lemonWave = new Artikel(202, "Lemon Wave",
			"Smartphone", 199.99);
		Artikel lemonSharp = new Artikel(203, "Lemon Sharp",
			"Smartphone", 249.99);
		Artikel lemonTop = new Artikel(301, "Lemon Top", "Tablet",
			499.99);
		Artikel lemonOptimum = new Artikel(302, "Lemon Optimum",
			"Tablet", 999.99);
	}

}
```

8.4.3 Kunden erzeugen

Nachdem wir unsere Artikel angelegt haben, wollen wir auch die folgenden Kunden der Lemonline AG mit den angegebenen Kundennummern und Adressen erzeugen:

- Geschaeftskunde "casarella": Name=Casarella, Kundennummer=1001, Adresse=Schmuckweiher 14 35394 Gießen, UstIdD=DE123456789
- Geschaeftskunde "orangeClub": Name=Orange Club, Kundennummer=1002, Adresse=Unter den Linden 5 10117 Berlin, UstID=DE987654321
- Privatkunde "rMaier": Name=Maier, Kundennummer=2001, Adresse=Licher Strasse 70 35394 Giessen, Vorname=Robert
- Privatkunde "tMueller": Name=Mueller, Kundennummer=2002, Adresse=Ludwigstraße 23 35390 Gießen, Vorname=Tina
- Privatkunde "jSchneider": Name=Schneider, Kundennummer=2003, Adresse=Senckenbergstraße 3 35390 Gießen, Vorname=Julius

```
package wbt08;

public class Webshop {

	public static void main(String[] args) {
		// Artikel erzeugen
		Geschaeftskunde casarella = new Geschaeftskunde("Casarella",
			1001, "Schmuckweiher 14 35394 Gießen", "DE123456789");
```

```
            Geschaeftskunde orangeClub = new Geschaeftskunde("Orange Club",
                  1002, "Unter den Linden 5 10117 Berlin", "DE987654321");
            Privatkunde rMaier = new Privatkunde("Maier", 2001,
                  "Licher Strasse 70 35394 Giessen", "Robert");
            Privatkunde tMueller = new Privatkunde("Mueller", 2002,
                  "Ludwigstraße 23 35390 Gießen", "Tina");
            Privatkunde jSchneider = new Privatkunde("Schneider", 2003,
                  "Senckenbergstraße 3 35390 Gießen", "Julius");
      }

}
```

8.4.4 Bestellungen aufgeben

Nachdem wir Artikel und Kunden in unserem Webshop erzeugt haben, können Kunden im Webshop bestellen. Beim Bestellvorgang können Kunden ihr Kundenkonto und die Artikeldetails mit der toString()-Methode von Kunden bzw. Artikeln aufrufen. Bitte erfassen Sie die folgenden Bestellungen in der Klasse *Webshop*:

- casarella: 3 Lemon Wave, 5 Lemon Top
- rMaier: 3 Lemon Classic
- orangeClub: 10 Lemon Sharp, 15 Lemon Optimum
- tMueller: 2 Lemon Star, 1 Lemon Five
- jSchneider: 1 Lemon Light

```
package wbt08;

public class Webshop {

      public static void main(String[] args) {
            // Artikel erzeugen // Kunden erzeugen
            System.out.println(casarella.toString());
            System.out.println(lemonWave.toString());
            System.out.println(lemonTop.toString());
            casarella.bestellen(lemonWave, 3, lemonTop, 5);

            System.out.println(rMaier.toString());
            System.out.println(lemonClassic.toString());
            rMaier.bestellen(lemonClassic, 3);

            System.out.println(orangeClub.toString());
            System.out.println(lemonSharp.toString());
            System.out.println(lemonOptimum.toString());
            orangeClub.bestellen(lemonSharp, 10, lemonOptimum, 15);

            System.out.println(tMueller.toString());
            System.out.println(lemonStar.toString());
            System.out.println(lemonFive.toString());
            tMueller.bestellen(lemonStar, 2, lemonFive, 1);
```

```
            System.out.println(jSchneider.toString());
            System.out.println(lemonLight);
            jSchneider.bestellen(lemonLight, 1);
      }

}
```

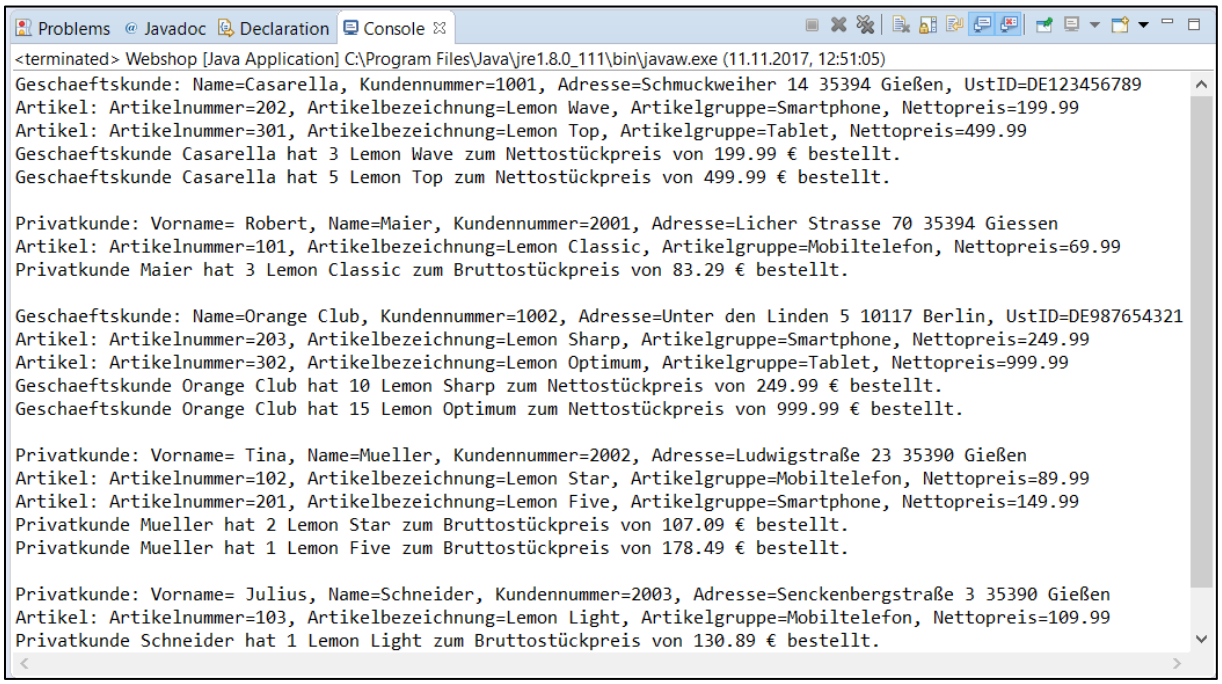


Abb. 73: Methoden „toString“ und „bestellen“ ausführen

8.4.5 Artikel anpassen

Horst Schäfer: „Da unsere Vertriebs-Abteilung regelmäßig Rabattaktionen für unsere Artikel startet, wollen wir als nächstes den Preis für das Mobiltelefon Lemon Classic ("lemonClassic") von 69,99 € auf 49,99 € reduzieren. Bitte lassen Sie sich dafür den aktuellen Preis des Lemon Classic mit der getter-Methode anzeigen. Danach können Sie mit setNettopreis den Preis auf 49,99 € reduzieren. Bitte überprüfen Sie Ihre Eingabe mit der entsprechenden getter-Methode.“

```
package wbt08;

public class Webshop {

      public static void main(String[] args) {
            // Artikel erzeugen // Kunden erzeugen
            System.out.println(lemonClassic.getNettopreis());
            lemonClassic.setNettopreis(49.99);
            System.out.println(lemonClassic.getNettopreis());
      }

}
```

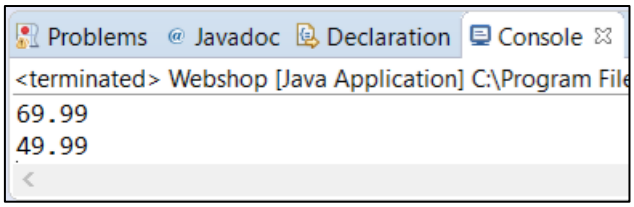


Abb. 74: Methoden „getNettopreis“ und „setNettopreis“ ausführen

8.4.6 Kunden anpassen

Horst Schäfer: „Zu Testzwecken wollen wir jetzt das Anpassen von Kundendaten überprüfen. Wir wollen die Adresse unseres Privatkunden Julius Schneider ("jSchneider") anpassen. Er wohnt ab jetzt in der Otto-Behaghel-Straße 27 in 35394 Gießen. Bitte lassen Sie sich die aktuelle Adresse des Kunden Julius Schneider mit der getter-Methode anzeigen. Verändern Sie danach die Adresse mit der setter-Methode auf Otto-Behaghel-Straße 27 35394 Gießen. Bitte überprüfen Sie Ihre Eingabe mit der getter-Methode.“

```
package wbt08;

public class Webshop {

	public static void main(String[] args) {
		// Artikel erzeugen // Kunden erzeugen
		System.out.println(jSchneider.getAdresse());
		jSchneider.setAdresse("Otto-Behaghel-Straße 27 35394 Gießen");
		System.out.println(jSchneider.getAdresse());
	}

}
```

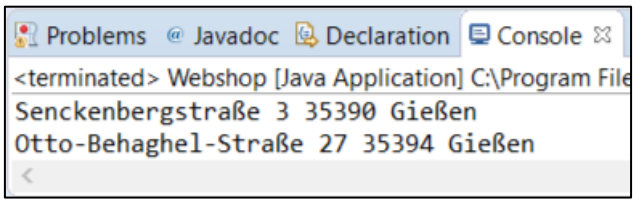


Abb. 75: Methoden „getAdresse“ und „setAdresse“ ausführen

8.4.7 Vertragsangebot

Horst Schäfer: „Frau Schmitt, wir sind am Ende Ihres Praktikums angekommen. Im Rahmen Ihres Praktikums haben Sie an einer Schulung zur objektorientierten Programmierung mit Java und Eclipse teilgenommen und einen Webshop für die Lemonline AG prototypisch programmiert. Ich möchte mich bei Ihnen für die gute Zusammenarbeit herzlich bedanken. Sie haben in den vergangenen Wochen sehr viel gelernt und sich hervorragend in das Team der IT- Abteilung

der Lemonline AG eingefügt. Nach einer kurzen Rücksprache mit unserem Vorstand freue ich mich, Ihnen mitteilen zu dürfen, dass wir Ihnen eine Anstellung als Werkstudentin bei der Lemonline AG anbieten können. Wenn Sie das Stellenangebot annehmen, werden Sie weiter auf Basis von UML-Klassendiagrammen objektorientierte Java-Programme mit Eclipse erstellen. Bitte teilen Sie uns Ihre Entscheidung zeitnah mit.“

8.4.8 Lessons learned

Horst Schäfer: „In der heutigen Schulung haben Sie die mit den Konzepten der Objektorientierung den Webshop für die Lemonline AG prototypisch programmiert. Sie haben gelernt, dass wir Objekte mithilfe von UML-Anwendungsfalldiagrammen und UML-Klassendiagrammen planen müssen, bevor wir den Java-Quellcode schreiben. Im Rahmen der WBT-Serie "Programmierung mit Java - Teil 2" haben Sie Ihr Wissen aus der WBT-Serie "Programmierung mit Java -Teil 1" vertieft. Zudem haben Sie einen umfassenden Einblick in die objektorientierte Java-Programmierung mit der IDE Eclipse erhalten.“

Literaturverzeichnis

1. **Ackermann, Philip:** Schrödinger programmiert Java - Das etwas andere Fachbuch, 2. Auflage, Bonn: Galileo Press 2017.
2. **Balzert, Heide:** Objektorientierung in 7 Tagen – Vom UML-Modell zur fertigen Web-An-wendung, Heidelberg; Berlin: Spektrum Akademischer Verlag 2000.
3. **Dornberg, Rolf; Telesko, Rainer:** Java-Training zur Objektorientierten Programmierung, München: Oldenbourg Wissenschaftsverlag GmbH 2010.
4. **Goll, Joachim; Heinisch, Cornelia:** Java als erste Programmiersprache - Grundkurs für Hochschulen, 8., überarbeitete Auflage, Wiesbaden: Springer Vieweg 2016.
5. **Günster, Kai:** Einführung in Java, Bonn: Rheinwerk Verlag 2015.
6. **Heinisch, Cornelia; Müller-Hofmann, Frank; Goll, Joachim:** Java als erste Programmiersprache – Vom Einsteiger zum Profi, 6., überarbeitete Auflage, Wiesbaden: Springer Vieweg 2011.
7. **Kottmair, Florian:** Java für Einsteiger: Einführung in die Programmierung mit Java ohne Vorkenntnisse, Berlin: epubli GmbH 2014.
8. **Lahres, Bernhard; Raýman, Gregor; Strich, Stefan:** Objektorientierte Programmierung - das umfassende Handbuch, 3., aktualisierte und erweiterte Auflage, Bonn: Rheinwerk Verlag 2016.
9. **Object Management Group (Hrsg.):** Unified Modeling Language™ (UML®), Online im Internet: http://www.omg.org/spec/UML/, 23.09.2017.
10. **Poetzsch-Heffter, Arnd:** Konzepte objektorientierter Programmierung: mit einer Einführung in Java, 2., überarbeitete Auflage, Berlin; Heidelberg: Springer 2009.
11. **Ullenboom, Christian:** Java ist auch eine Insel – Einführung, Ausbildung, Praxis, 13., aktualisierte und überarbeitete Auflage, Bonn: Rheinwerk Verlag 2017.
12. **Wolmeringer, Gottfried; Klein, Thorsten:** Profikurs Eclipse 3, 2., verbesserte und erweiterte Auflage, Wiesbaden: Springer Vieweg 2006.

Impressum

Reihe: **Arbeitspapiere Wirtschaftsinformatik** (ISSN 1613-6667)

Bezug: https://wi.uni-giessen.de

Herausgeber: Prof. Dr. Axel Schwickert
Prof. Dr. Bernhard Ostheimer

c/o Professur BWL – Wirtschaftsinformatik
Justus-Liebig-Universität Gießen
Fachbereich Wirtschaftswissenschaften
Licher Straße 70
D – 35394 Gießen
Telefon (0 64 1) 99-22611
Telefax (0 64 1) 99-22619
eMail: Axel.Schwickert@wirtschaft.uni-giessen.de
https://wi.uni-giessen.de

Ziele: Die Arbeitspapiere dieser Reihe sollen konsistente Überblicke zu den Grundlagen der Wirtschaftsinformatik geben und sich mit speziellen Themenbereichen tiefergehend befassen. Ziel ist die verständliche Vermittlung theoretischer Grundlagen und deren Transfer in praxisorientiertes Wissen.

Zielgruppen: Als Zielgruppen sehen wir Forschende, Lehrende und Lernende in der Disziplin Wirtschaftsinformatik sowie das IT-Management und Praktiker in Unternehmen.

Quellen: Die Arbeitspapiere entstehen aus Forschungs-, Abschluss-, Studien- und Projektarbeiten sowie Begleitmaterialien zu Lehr-, Vortrags- und Kolloquiumsveranstaltungen der Professur BWL – Wirtschaftsinformatik, Prof. Dr. Axel Schwickert, Justus-Liebig-Universität Gießen sowie der Professur für Wirtschaftsinformatik, insbes. medienorientierte Wirtschaftsinformatik, Prof. Dr. Bernhard Ostheimer, Fachbereich Wirtschaft, Hochschule Mainz.

Hinweise: Wir nehmen Ihre Anregungen zu den Arbeitspapieren aufmerksam zur Kenntnis und werden uns auf Wunsch mit Ihnen in Verbindung setzen.

Falls Sie selbst ein Arbeitspapier in der Reihe veröffentlichen möchten, nehmen Sie bitte mit einem der Herausgeber unter obiger Adresse Kontakt auf.

Informationen über die bisher erschienenen Arbeitspapiere dieser Reihe erhalten Sie unter der Web-Adresse https://wi.uni-giessen.de/