Programmiersprache

Java

2021 / Teil 2

Detlef Wilkening
www.wilkening-online.de

Programmiersprache Java

4 Praxis

Neben einer IDE (z.B. Eclipse, Netbeans, IntelliJ), in der man typischerweise entwickelt, ist auch das JDK notwendig.

Hinweis: die Abbildungen und Pfade beziehen sich auf eine ältere Java Version 1.8.0u40. Benutzen Sie die aktuellste Version. Vom Prinzip ist natürlich alles identisch.

4.1 JDK

Mit Installation des Java-Development-Toolkits (JDK) stehen Ihnen auf der Kommandozeile direkt die Java Entwicklungswerkzeuge zur Verfügung. Für unsere einfache Einführung schauen wir uns nur die Java-Virtual-Machine (JVM) und den Java-Compiler (javac) an. Sie können die JVM mit „java“ aufgerufen werden. Mit der Option „-version“ z.B. gibt die JVM nur ihre Version aus und beendet sich dann direkt.

Abb. 4-1 : Installiertes JDK – Aufruf der JVM

Wollen Sie auch den Java-Compiler „javac“ und die anderen JDK-Tools direkt nutzen können, so müssen Sie z.B. unter Windows in der System-Steuerung den Pfad zu dem Bin-Verzeichnis Ihrer JDK-Installation der Umgebungs-Variablen „path“ hinzufügen. Bei einer „normalen“ Installation ist dies z.B. für das JDK 1.8.0_40 unter einer 64 Bit Windows Version: „C:\Program Files\Java\jdk1.8.0_40\bin“. Wenn Sie dies gemacht haben, dann können Sie auch den Java-Compiler „javac“ direkt auf der Kommandozeile aufrufen – z.B. auch mit der Option „-version“.

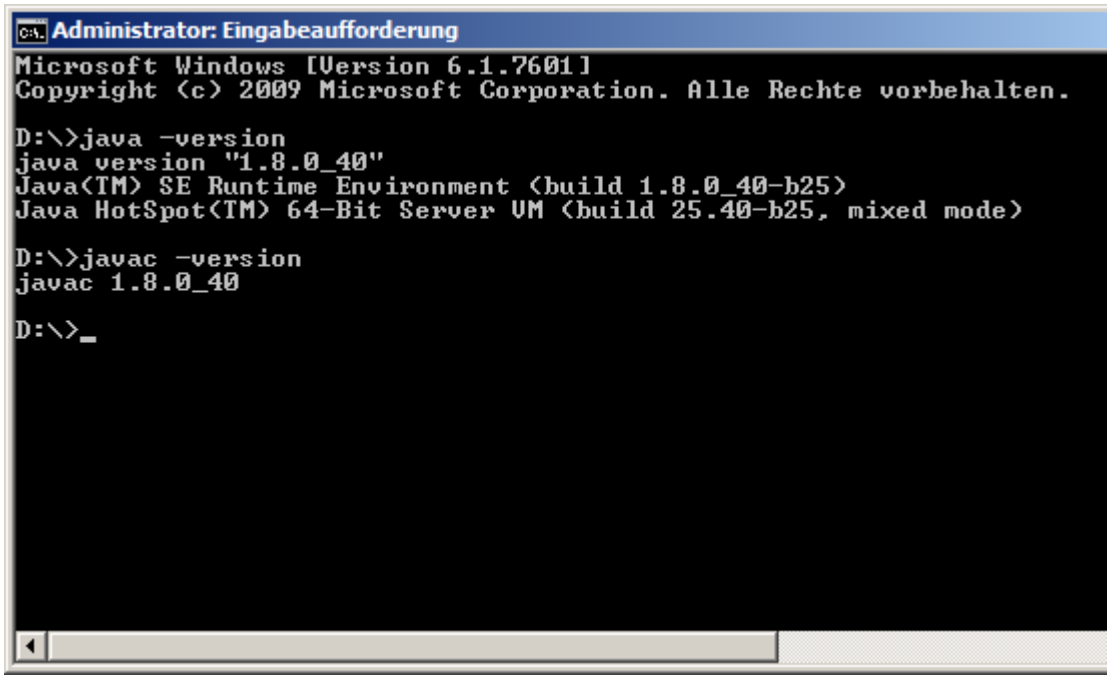

Abb. 4-2 : Aufruf des Java-Compilers nach gesetzter Path Umgebungs-Variable

Haben Sie das JDK Bin-Verzeichnis nicht in den Pfad aufgenommen, dann müssen Sie beim Aufruf den kompletten Pfad-Namen inkl. Extension mitangeben:

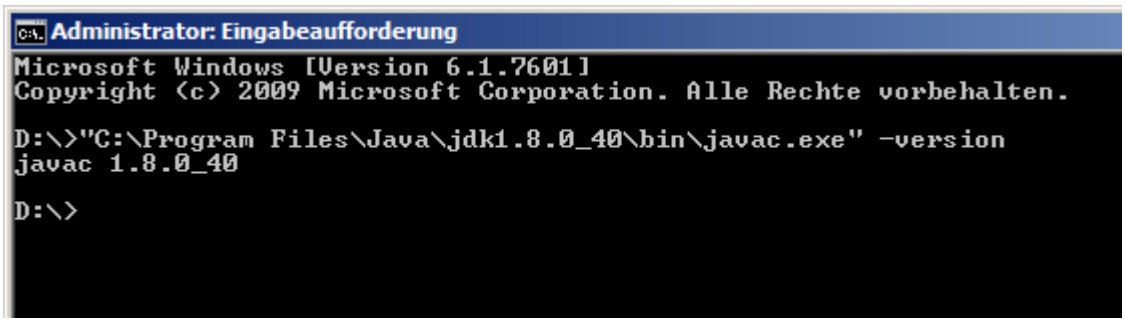

Abb. 4-3 : Aufruf des Java-Compilers mit vollem Pfad

4.2 Source Organisation

Bevor wir unsere ersten Programme übersetzen und laufen lassen, noch einmal die folgenden sehr wichtigen Hinweise:

- Die gesamte programmierte Logik in einem Java Programm spielt sich in Klassen ab.
- Jeder Java Quelltext **muss genau eine** public Klasse (oder Interface) enthalten.
- Der Name der Klasse **muss** dem Namen der Quelldatei entsprechen (inkl. Groß- und Kleinschreibung). Die Datei **muss** zusätzlich die Extension „.java“ haben.
  - Beispiel: Die public Klasse „Pop3Protocol“ muss zwingend in einer Datei „Pop3Protocol.java“ stehen.
- Liegt die Klasse in einem Package, so muss die Datei in einem Verzeichnis mit dem entsprechenden Package Namen liegen (auch hier gilt natürlich wieder Gross- und Kleinschreibung).
  - Beispiel: Liegt die public Klasse „TestCase“ in einem Package „applicationlogic“, so muss die Datei „TestCase.java“ zwingend in einem Verzeichnis „applicationlogic“ liegen.
  - Beispiel: Liegt die public Klasse „SeqNode“ in den Packages „db“ und „implementation“ (d.h. das Package „db“ enthält das Package „implementation“), so muss die Datei „SeqNode.java“ in einem Verzeichnis „implementation“ liegen, und dieses wiederum in einem Verzeichnis „db“.
- Der Grund für die Korrelation von Klassen- und Datei-Namen, bzw. von Package- und Verzeichnis-Namen wird in Kapitel 4.4 erklärt.

Die Korrelation von Klassen- zu Datei-Namen, und Package- zu Verzeichnis-Namen wurde zwar schon früher erwähnt, aber die Erfahrung zeigt, dass dies ein ganz typischer Anfänger-Fehler ist. Darum sollte es hier noch mal erwähnt sein. Passen Sie also darauf auf. In den letzten Jahren waren sicher die Hälfte der Probleme in den ersten Praktikumswochen auf diesen Fehler zurückzuführen.

4.3 Benutzung der JDK Entwicklungs-Tools

Für die Beispiele hier wird davon ausgegangen, dass unter Windows gearbeitet wird, das aktuelle JDK 1.8 installiert ist, und für z.B. den Java-Compiler das JDK Bin-Verzeichnis in die Path Umgebungs-Variable eingetragen wurde.

4.3.1 Java mit Klassen ohne Packages

Liegen die Java Klassen in keinem Package, so ist die Benutzung der Kommandozeilen-Tools sehr einfach. Schauen wir uns die die Schritte zur Verarbeitung und Ausführung des „Hallo-Welt“ Beispiels an.

Achtung – Klassen, die in keinem Package liegen, sollten die absolute Ausnahme sein. Sie sind eine schnelle Lösung für kleine Test- oder Beispiel-Programme – von daher werden Sie solche viel in diesem Tutorial finden. Für mehr sollte man sie aber nicht nutzen. Später – wenn Sie reale Programme schreiben – sollten Sie für alle ihre Programme eine sinnvolle Datei-Organisation anlegen, die sich dann in den Packages und Verzeichnissen widerspiegelt.

4.3.1.1 „Hallo Welt“ Beispiel

Erstellen Sie mit einem Editor den Quelltext des „Hallo-Welt“ Java Programms, diesmal aber mit dem Klassen-Namen „HalloWelt“, und speichern Sie ihn unter dem Namen „HalloWelt.java“ ab. Bei mir liegt die Datei z.B. direkt unter „E:\java“.

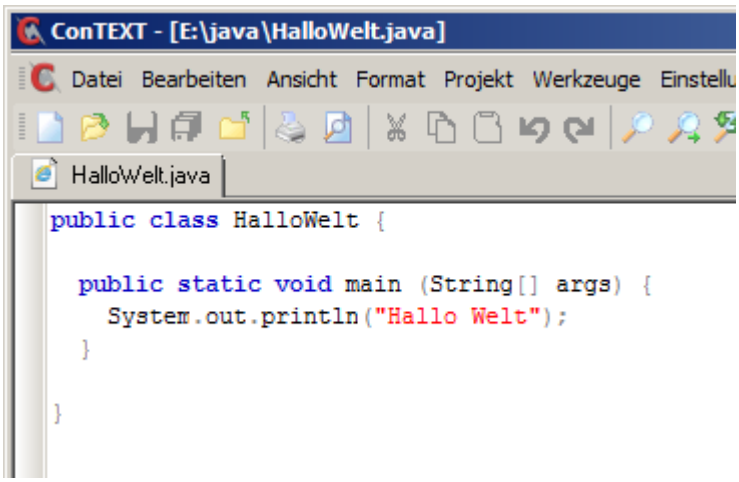

Abb. 4-4 : „Hallo Welt“ Quelltext im Editor (Datei „e:\java\HalloWelt.java“)

Achtung – passen Sie auf, dass die Klasse genauso wie die Datei heißt (bis auf die zusätzliche Extension „.java“) – die Sprache Java fordert dies.

Öffnen Sie jetzt eine Kommandozeile für dieses Verzeichnis.

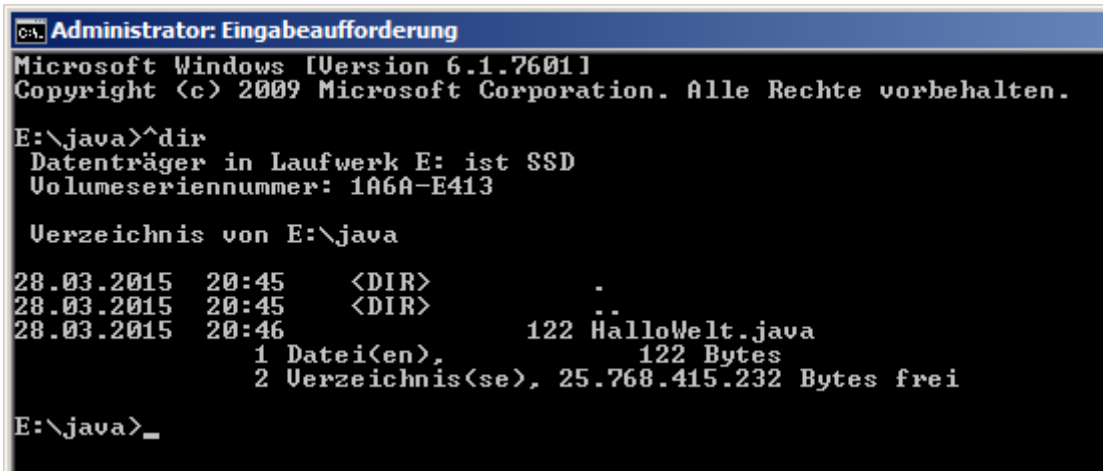

Abb. 4-5 : Kommandozeile für das Verzeichnis „e:\java“

Rufen Sie den Java Compiler „javac“ mit Angabe der zu übersetzenden Datei auf. In Abhängigkeit von ihrer Konfiguration müssen Sie entweder den Pfad zum Compiler mit angeben, oder können ihn weglassen. Bei mir ist der Pfad zu den JDK Entwicklungs-Tools in

der Pfad-Umgebungsvariablen enthalten – ich brauche den Pfad also nicht anzugeben:

```
> javac HalloWelt.java
```

Der Java Compiler compiliert die angegebene Datei und erzeugt den entsprechenden Byte-Code (Datei „HalloWelt.class“) im aktuellen Verzeichnis.

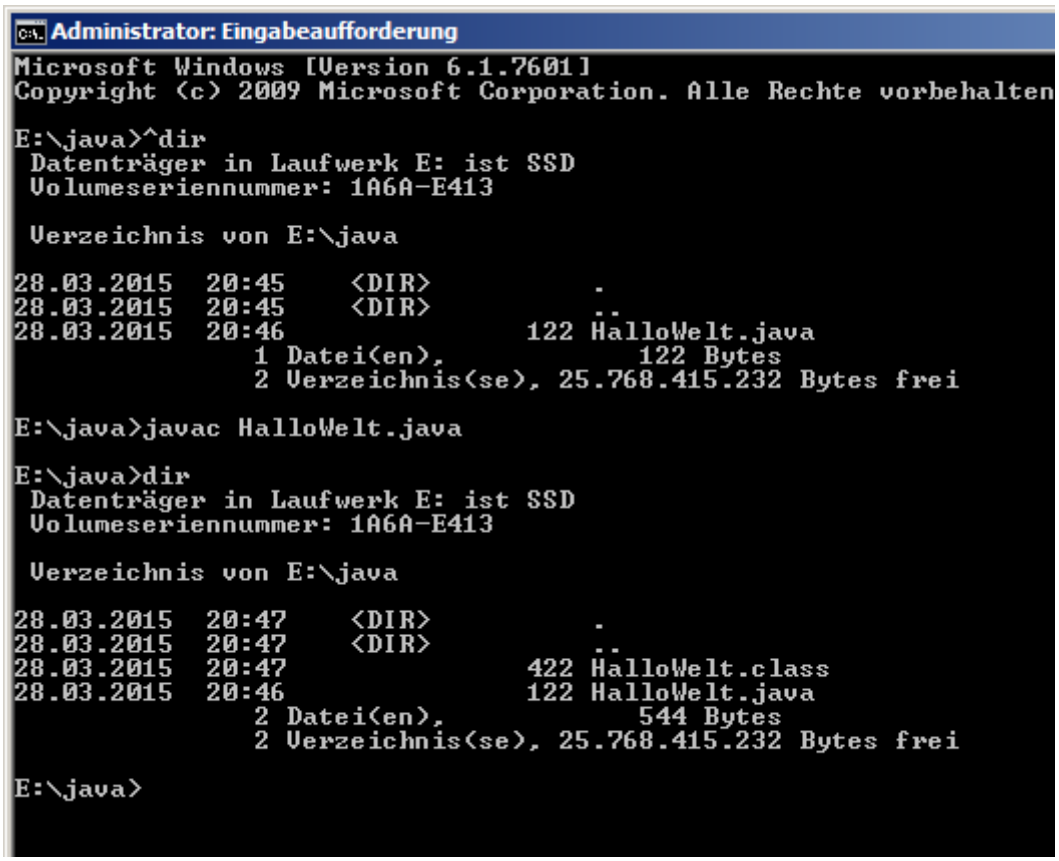

Abb. 4-6 : Der Java Compiler erzeugt den Byte-Code des „Hallo Welt“ Beispiels

Diesen müssen wir jetzt in der virtuellen Java Maschine (JVM) ausführen. Dazu rufen wir die virtuelle Maschine „java“ auf, und übergeben den **vollständigen Namen der Klasse**.

```
> java HalloWelt
```

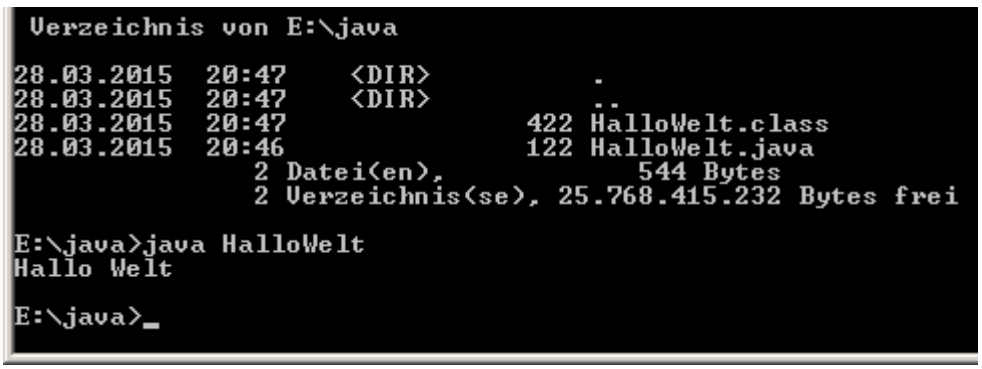

Abb. 4-7 : Die virtuelle Java Maschine (JVM) führt das „Hallo Welt“ Beispiel aus

4.3.1.2 GUI Beispiel

Vergleichbar zum „Hallo-Welt“ Beispiel läuft Handling des GUI Beispiels ab.

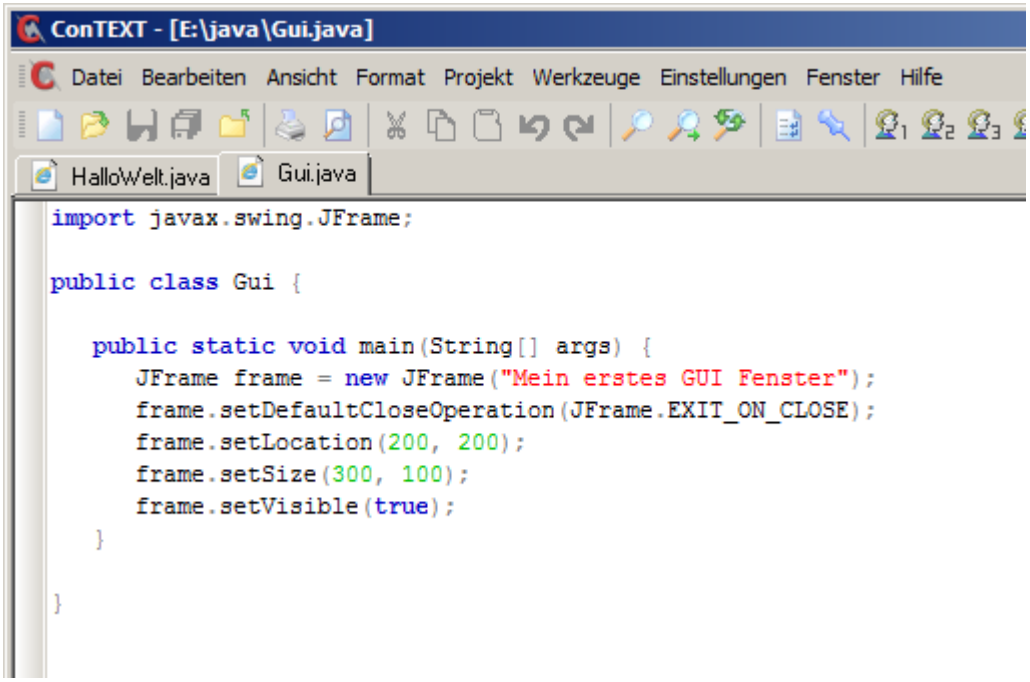

Abb. 4-8 : GUI Quelltext im Editor (Datei „e:\java\Gui.java“)

```
 Verzeichnis von E:\java

28.03.2015  20:51    <DIR>          .
28.03.2015  20:51    <DIR>          ..
28.03.2015  20:51               329 Gui.java
28.03.2015  20:47               422 HalloWelt.class
28.03.2015  20:46               122 HalloWelt.java
               3 Datei(en),            873 Bytes
               2 Verzeichnis(se), 25.768.415.232 Bytes frei

E:\java>javac Gui.java

E:\java>dir
 Datenträger in Laufwerk E: ist SSD
 Volumeseriennummer: 1A6A-E413

 Verzeichnis von E:\java

28.03.2015  20:51    <DIR>          .
28.03.2015  20:51    <DIR>          ..
28.03.2015  20:51               523 Gui.class
28.03.2015  20:51               329 Gui.java
28.03.2015  20:47               422 HalloWelt.class
28.03.2015  20:46               122 HalloWelt.java
               4 Datei(en),          1.396 Bytes
               2 Verzeichnis(se), 25.768.415.232 Bytes frei

E:\java>
```

Abb. 4-9 : Der Java Compiler erzeugt den Byte-Code des GUI Beispiels

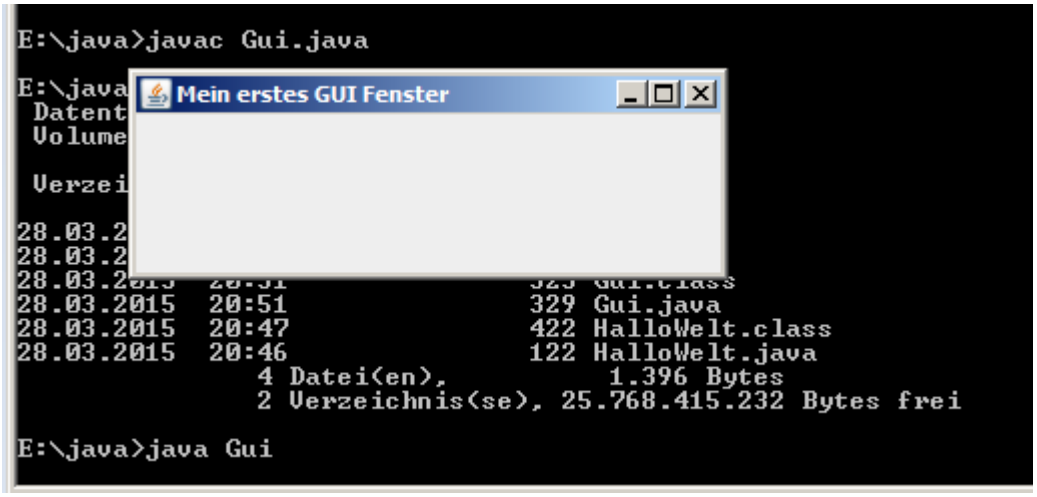

Abb. 4-10 : Die virtuelle Java Maschine (JVM) führt das GUI Beispiel aus

4.3.1.3 Mehrere Klassen gleichzeitig compilieren

Wollen Sie mehrere Dateien auf einmal übersetzen, so können Sie den Wildcard „*“ im Datei-Namen benutzen. Löschen Sie z.B. die beiden gerade erzeugten Class-Dateien, und compilieren Sie unsere beiden Beispiel-Quelltexte „HalloWelt.java“ und „Gui.java“ in einem Rutsch neu:

```
> javac *.java
```

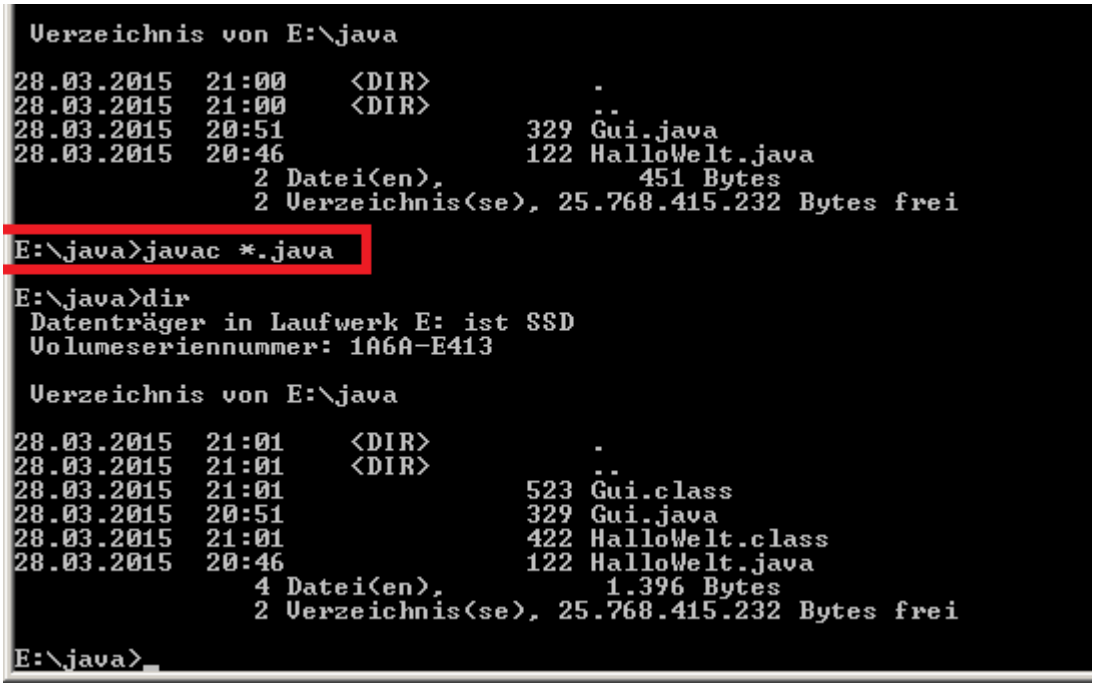

Abb. 4-11 : Der Java Compiler erzeugt den Byte-Code aller Klassen im Verzeichnis

4.3.2 Java mit Klassen in Packages

Liegen die Klassen in Packages, so wird das ganze etwas aufwändiger, aber nicht wirklich kompliziert. Um das zu lernen, schreiben wir ein Beispiel ähnlich dem „Hallo-Welt“ von eben, nur dass die Klasse jetzt in den Packages „aussen.mitte.innen“ liegt. Natürlich legen wir die Datei korrekt in die Verzeichnis-Struktur „aussen\mitte\innen“.

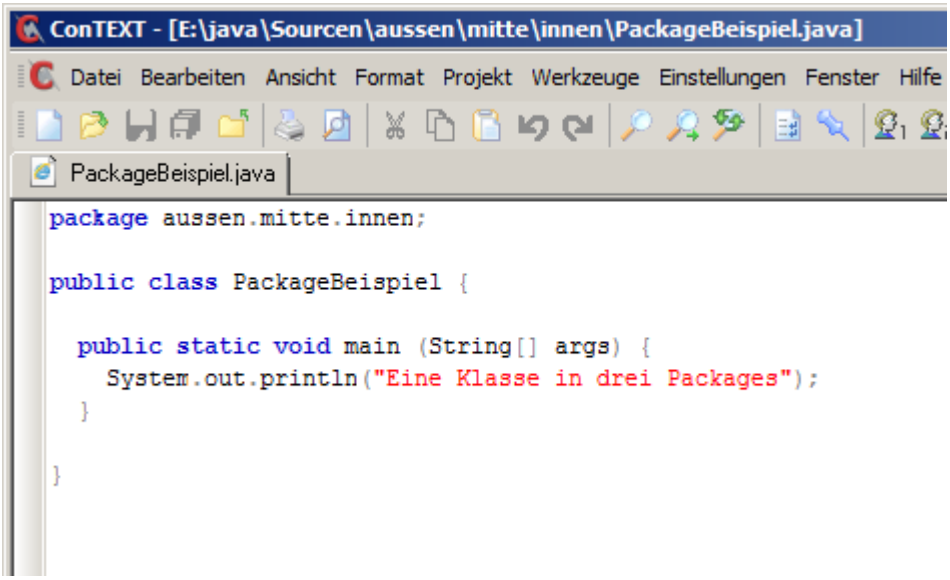

Abb. 4-12 : Package Quelltext im Editor (Datei „PackageBeispiel.java“)

Damit das Beispiel auch später noch nutzbar ist, wenn wir die Source- und Class-Path Optionen erklären (siehe Kapitel 4.3.2.3 und Kapitel 4.4), legen wir die Verzeichnis-Struktur „aussen\mitte\innen“ nicht direkt in „e:\java“ an, sondern „betten“ sie noch zusätzlich in ein Zwischen-Verzeichnis „Sourcen“ ein. So sieht unsere Struktur dann aus:

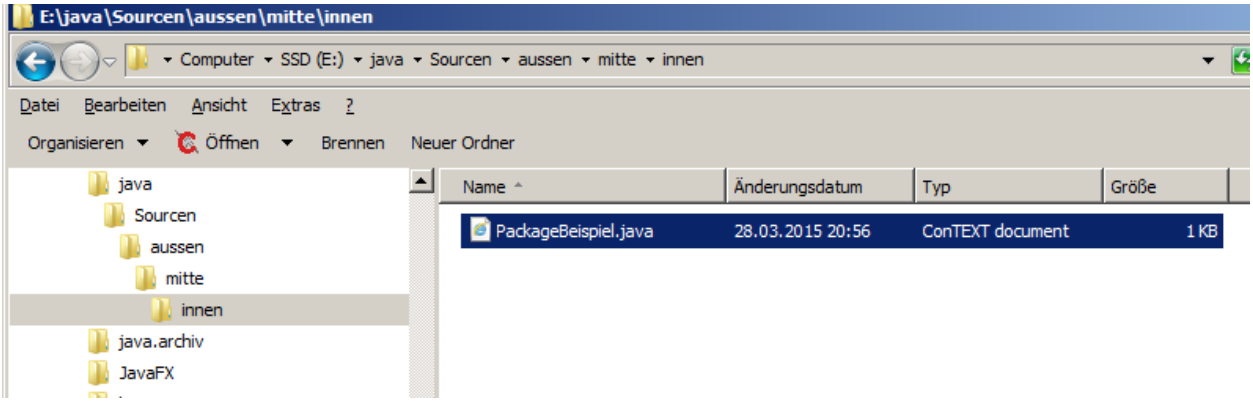

Abb. 4-13 : Verzeichnis-Struktur für das Package Beispiel

4.3.2.1 Einfache, aber falsche Vorgehensweise

Wenn Sie jetzt einfach die Vorgehensweise von eben übertragen, erleben Sie eine kleine Überraschung. Aber warum nicht – probieren wir es einfach mal aus:

- Wir wechseln mit der Konsole in das Verzeichnis „innen“
- Wir rufen den Java Compiler auf
- Und wir starten das Programm...

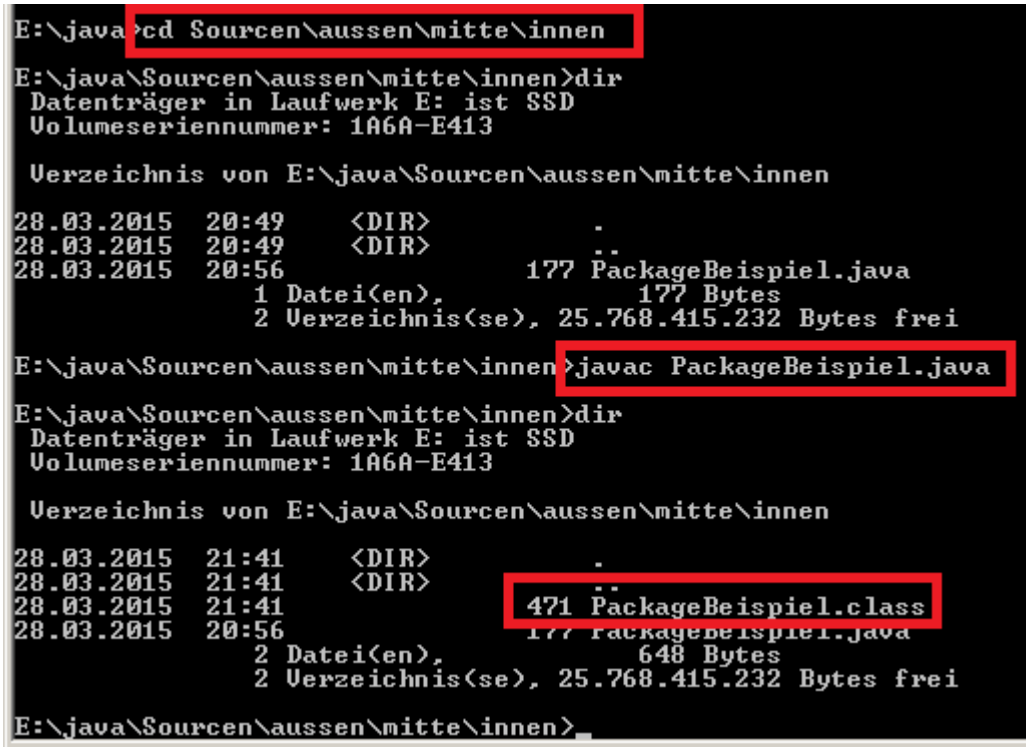

Abb. 4-14 : Der Java Compiler erzeugt den Byte-Code des Package Beispiels

Wie wir sehen, compilierte der Compiler den Quelltext ganz problemlos, und der Byte-Code liegt damit vor. Also starten wir das Programm mal:

Abb. 4-15 : Fehler bei der Ausführung des Package Beispiels

Was ist das? Unser Programm startet nicht – stattdessen meldet uns die JVM einen Fehler. Um das Problem vollständig zu verstehen, fehlt uns noch etwas Wissen. Im Augenblick soll die Bemerkung reichen, dass wir versucht haben, den Compiler und vor allem die JVM zu

betrügen. Beim Starten des Programms erzählen wir der JVM von einer Klasse „Beispiel“ ohne Packages, die aber in Packages liegt.

Belassen wir es dabei – keine Details, die kommen später – lernen wir lieber, wie man es richtig macht. Aber vorher löschen Sie den fehlerhaft erzeugten Byte-Code „PackageBeispiel.class“, bevor er uns noch mehr Ärger macht.

4.3.2.2 So ist es richtig

Statt in das Verzeichnis mit dem Java Quelltext zu wechseln, müssen Sie bei der Übersetzung das Wurzel-Verzeichnis für die Quelltexte angeben. Das Wurzel-Verzeichnis ist quasi das Verzeichnis, das das äußere Package enthält, bei uns also „e:\java\Sourcen“.

Wechsel wir also zurück in unser Java-Verzeichnis. Wären wir dort also einfach mit unserer Konsole geblieben, hätten wir uns viel vergebene Arbeit sparen können. Zurück in „e:\java“ werfen wir den Compiler an, und geben jetzt beim Compiler-Aufruf zwei weitere Optionen an:

- Die Option „-sourcepath“ gibt das (oder die) Wurzel-Verzeichnis Ihrer zu übersetzenden Klassen an – in unserem Beispiel „Sourcen“ als relative Pfad-Angabe oder „e:\java\Sourcen“ als absolute Pfad-Angabe.
- Die zu übersetzende Java-Datei (oder die Java-Dateien) müssen wir jetzt natürlich inkl. Pfad angeben.

```
> javac -sourcepath Sourcen Sourcen\aussen\mitte\innen\PackageBeispiel.java
```

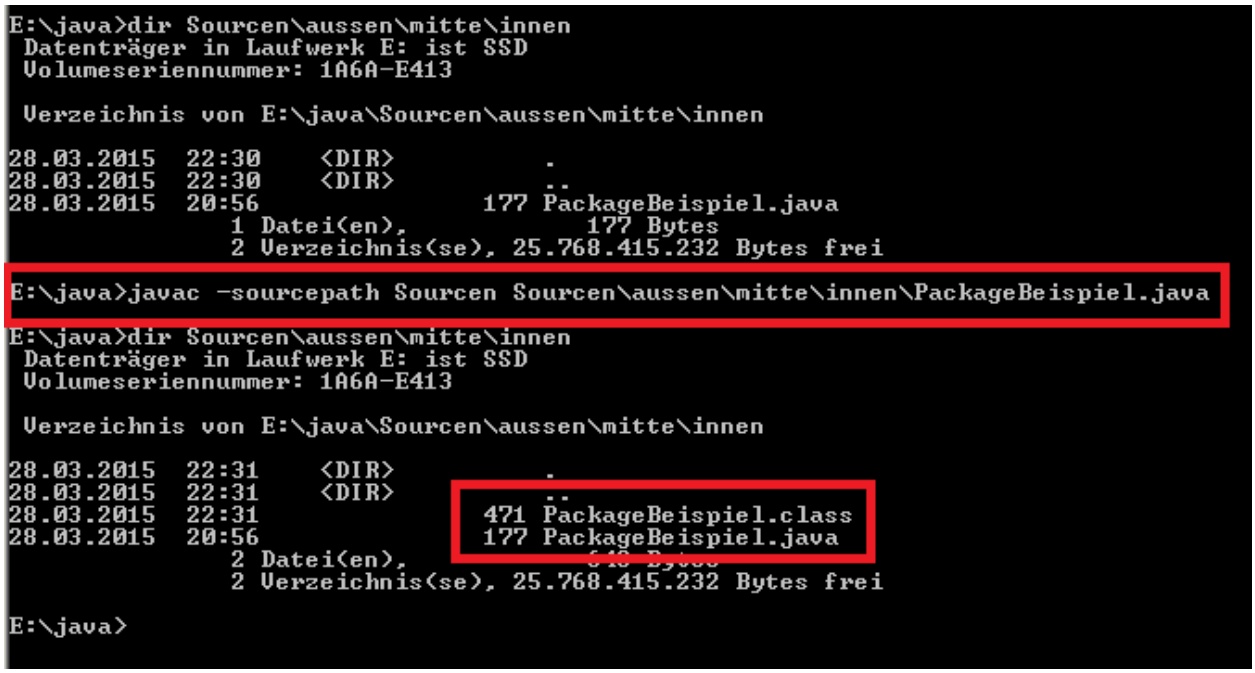

Abb. 4-16 : Der Java Compiler erzeugt den Byte-Code mit „sourcepath“ Option

Hinweise:

- Natürlich kann man auch hierbei Wildcards benutzen, um alle Dateien gleichzeitig zu

compilieren.

- Stehen Sie während des Compilierens im Wurzel-Verzeichnis Ihres Programms, so geben Sie als Soure-Path einfach nur den Punkt „.“ an, der ja für das aktuelle Verzeichnis steht.

Für die Ausführung wechseln wir nun in das Wurzel-Verzeichnis unseres Programms, d.h. in „e:\java\Sourcen“. Wir werden gleich noch lernen, wie man das Programm aus einem beliebigen Verzeichnis ausführt – aber erstmal einfach:

Beim Aufruf der JVM muss der **vollständige** Klassen-Name der Main-Klasse mitgegeben werden. Das war auch eben schon so, aber eben lag die Klasse nicht in einem Package – von daher war der einfache Klassen-Name auch gleichzeitig der vollständige Klassen-Name. Jetzt, mit Packages, ist der vollständige Klassen-Name der Klassen-Name inkl. Package-Namen – getrennt durch Punkte, d.h.: „aussen.mitte.innen.PackageBeispiel“.

```
> cd Sourcen
> java aussen.mitte.innen.PackageBeispiel
```

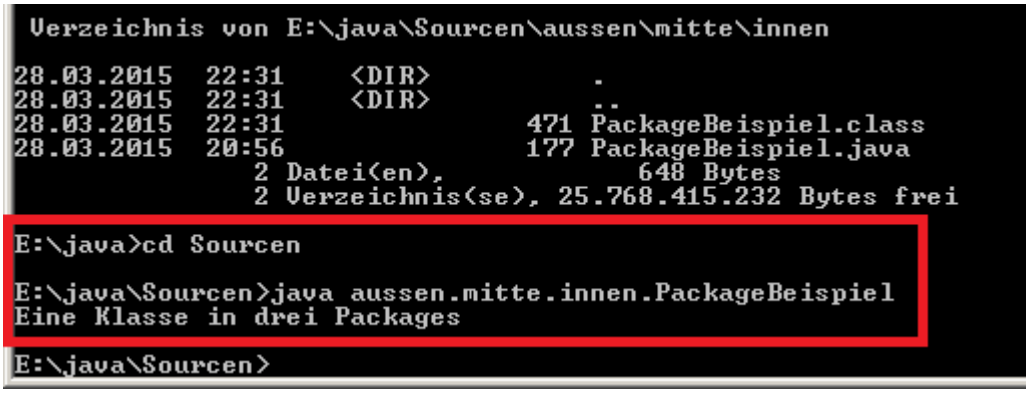

Abb. 4-17 : Korrekte Ausführung des Package Beispiels

4.3.2.3 Und was, wenn man nicht im Wurzel-Verzeichnis steht?

Lassen Sie uns wieder zurück in unser Java-Verzeichnis „e:\java“ wechseln. Nun stehen wir nicht mehr im Wurzel-Verzeichnis unseres Programms – wie starten wir es denn nun? Wenn wir hier die JVM mit der Main-Klasse aufrufen bekommen wir nur einen Fehler:

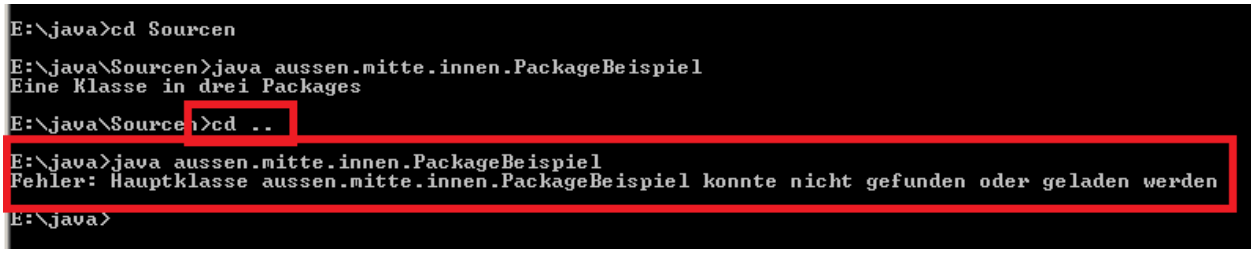

Abb. 4-18 : Fehler bei Ausführung außerhalb des Wurzel-Verzeichnisses

Das war auch zu erwarten. Woher soll die JVM wissen, wo unser Programm auf der Platte liegt.

Wir stehen quasi in einem beliebigen Verzeichnis – und in einem beliebigen anderen Verzeichnis liegt unser Programm.

In diesem Fall, muß man – analog zur Option „-sourcepath“ beimm Compiler – nun der JVM das Wurzel-Verzeichnis angeben. Nur heißt die Option hier „-classpath“ oder kurz „-cp“, da sich die JVM nicht mehr für die Sourcen interessiert, sondern statt dessen den Klassen-Pfad für den Byte-Code kennen muss.

```
> java -cp Sourcen aussen.mitte.innen.PackageBeispiel
```

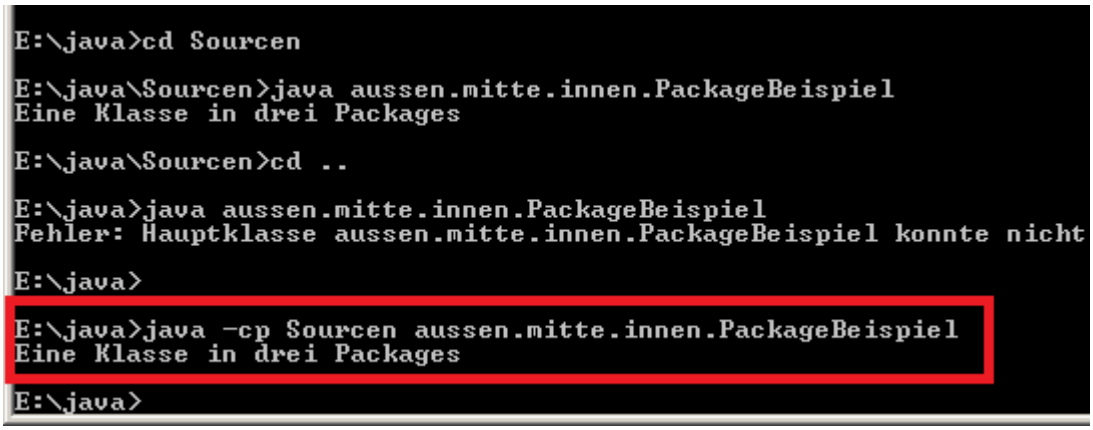

Abb. 4-19 : Korrekte Ausführung des Package Beispiels mit Class-Path Option „-cp“

Achtung – genauso wie die Sourcen definiert heißen und an genau definierten Stellen (relativ zum Source-Wurzel-Verzeichnis) liegen müssen, so muß auch der Byte-Code genau definiert heißen und relativ zu einer Klassen-Pfad-Wurzel genau definiert liegen. Also benennen Sie niemals Byte-Code um bzw. verschieben Sie ihn auch nicht, bzw. nur sehr bewußt.

4.3.3 Trennung von Sourcen und Byte-Code

Nun ist die Vermischung von Source-Code und Byte-Code im gleichen Verzeichnis keine gute Idee – oder sagen wir allgemein: die Vermischung von User-erzeugten und automatisch generierten Dateien. Schön wäre es, wenn der Java-Compiler den Byte-Code in ein extra Verzeichnis legen würde. Dies kann mit der Option „-directory“ oder kurz „-d“ erreicht werden.

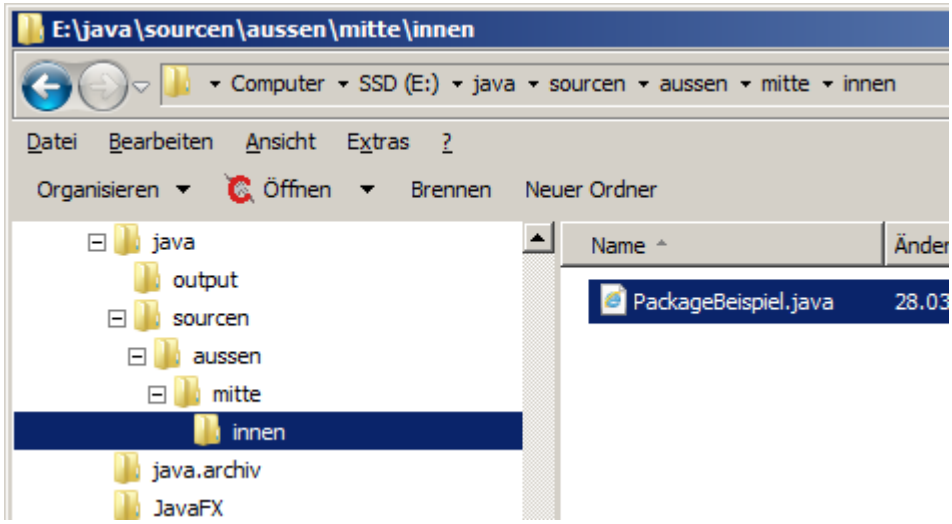

Abb. 4-20 : Ausgangs Struktur für Byte-Code eigene Verzeichnisse

Achtung - das Ausgabe Verzeichnis (hier „output“) muss existieren.

Beispielhaftes Compilieren aus dem Verzeichnis „e:\java“ heraus.

```
> javac -sourcepath sourcen -d output sourcen\aussen\mitte\innen\*.java
```

Abb. 4-21 : Der Java Compiler erzeugt den Byte-Code mit Trennung des Byte-Codes

Der Compiler erzeugt nun beim Compilieren nicht nur den Byte-Code, sondern auch die den Packages bzw. dem Source-Path entsprechende Verzeichnis-Struktur. D.h. im Ausgabe-Verzeichnis liegt nicht einfach die Byte-Code Datei „PackageBeispiel.class“. Statt dessen hat der Compiler auch die Verzeichnisse „aussen“, „mitte“ und „innen“ angelegt, und erst im Verzeichnis „innen“ liegt der Byte-Code.

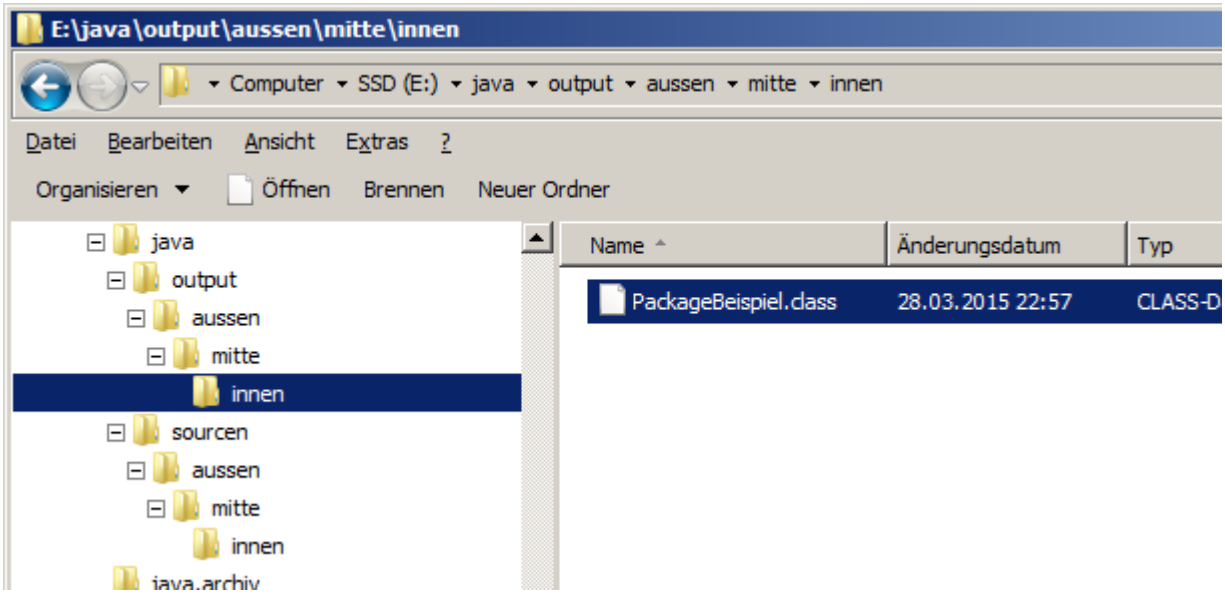

Abb. 4-22 : Erzeugte Verzeichnis Struktur und Byte-Code

Denken Sie jetzt beim Starten des Programms nur daran, dass Sie den Class-Path (hier z.B. „output“), und nicht den Source-Path (hier z.B. „sourcen“) angeben müssen. Die Sourcen und der Byte-Code liegen jetzt ja in zwei unterschiedlichen Verzeichnis-Strukturen.

```
> java -cp output aussen.mitte.innen.PackageBeispiel
```

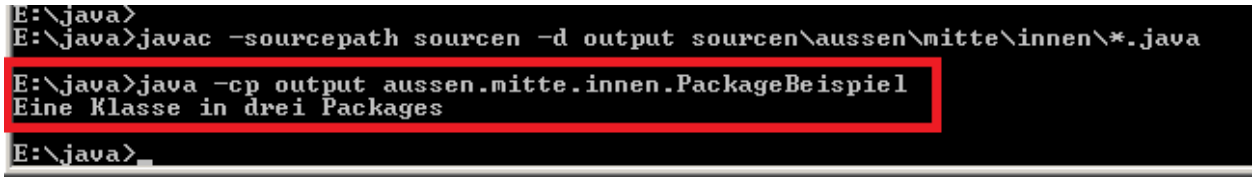

Abb. 4-23 : Korrekte Ausführung des Package Beispiels mit Trennung des Byte-Codes

Hinweis – wenn Sie so arbeiten, d.h. mit der Trennung des Source-Codes vom Byte-Code, dann müssen Sie immer nur Ihr Source-Verzeichnis sichern, und haben damit immer alles im Griff. Der Byte-Code läßt sich ja jederzeit wieder herstellen.

4.3.4 Kommandozeilen-Argumente

Sie können an ein Java Programm Argumente übergeben, indem Sie diese als letzte Argumente – d.h. nach dem Klassen-Namen – an die JVM übergeben. Unser Beispiel ist der Einfachheit halber ohne Packages, und lehnt sich an das alte Beispiel an. Achtung, das Beispiel soll mit zwei Kommandozeilen-Argument aufgerufen werden:

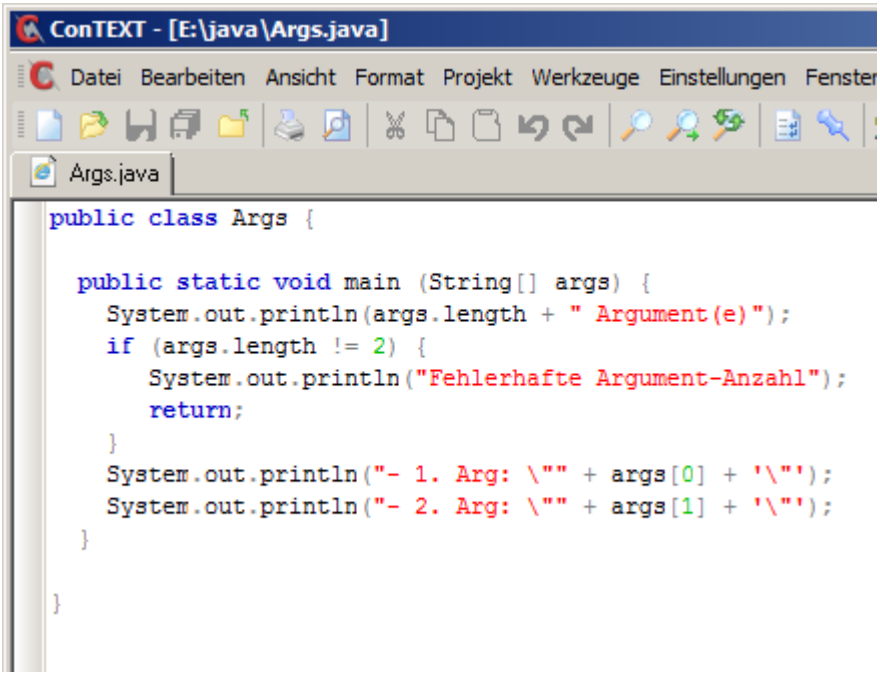

Abb. 4-24 : Kommandozeilen Argument Quelltext im Editor (Datei „e:\java\Args.java“)

```
> javac Args.java
> java Args
> java Args Hallo
> java Args Hallo Java
```

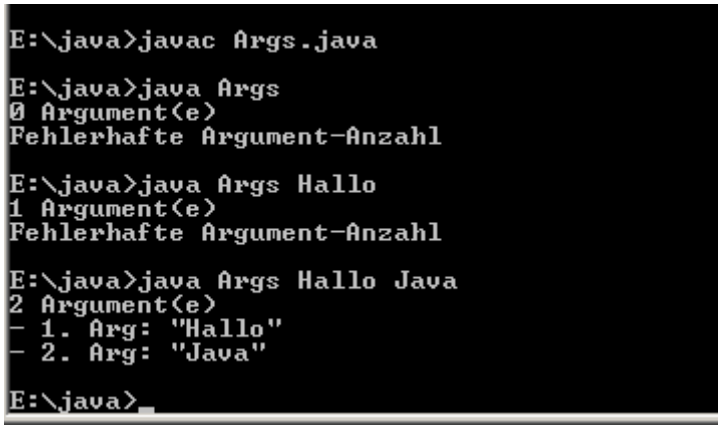

Abb. 4-25 : Compilation und Ausführung des Kommandozeilen Argumente Beispiels

4.4 Source-Path und Class-Path

4.4.1 Source-Path

Was sollen eigentlich beim Compilieren und Ausführen diese komischen Source- und Class-

Path Angaben. Schauen wir uns dafür mal den Source-Path beim Compilieren an, und nehmen wir an, wir haben in einem Projekt die zwei Klassen „Class1“ und „Class2“ in zwei parallel liegenden Packages „pack1“ und „pack2“.

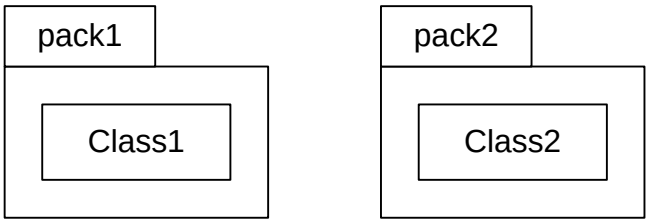

Abb. 4-26 : 2 Klassen in zwei parallelen Packages

Selbst wenn wir noch nicht viel über Klassen und Packages wissen, sollte uns klar sein, dass es passieren kann (und auch soll) dass eine Klasse die andere nutzt.

```
package pack1;

public class Class1 {
  private pack2.Class2 cl;   // Class1 nutzt Class2 aus dem Package pack2
}
```

Klar sollte uns auch sein, dass der Compiler überprüft, ob es die Klasse z.B. überhaupt gibt, ob „Class1“ sie benutzen darf, oder ob die Benutzung korrekt erfolgt (z.B. entsprechende Funktionen in der Klasse vorhanden sind).

Woher weiss der Compiler aber nun, wo die Klasse „pack2.Class2“ zu finden ist, um diese Dinge zu überprüfen? Hier kommen drei Dinge ins Spiel:

1) Eine Klasse heißt wie die Datei, in der sie steht.
2) Ein Package heißt wie das Verzeichnis, das es darstellt.
3) Der Source-Path zeigt auf die Wurzel(n) der Package-Struktur.

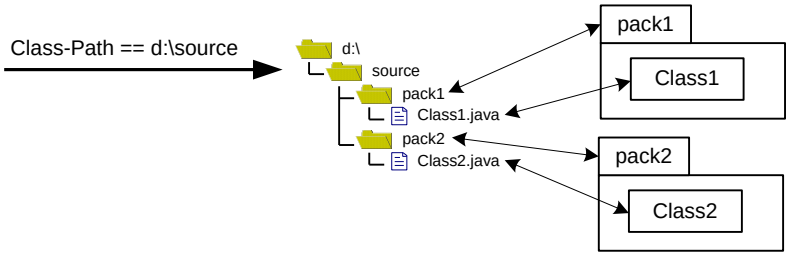

Abb. 4-27 : Klassen, Dateien, Packages, Verzeichnisse und der Source-Path

Der Compiler findet im Quelltext also die Referenzierung der Klasse „pack2.Class2“ und weiß sofort folgende Dinge:

1) Das Package heißt „pack2“ – also muss es im Source-Path Verzeichnis als der Wurzel aller Sourcen ein Verzeichnis mit Namen „pack2“ geben.
2) Die Klasse heißt „Class2“, also muss es in diesem Verzeichnis eine Datei „Class2.java“

geben, die die Klasse „Class2“ enthält und detailiert definiert.
Mit diesem Wissen kann er also die Definition der Klasse „Class2“ finden, sie einlesen, und alle notwendigen Überprüfungen durchführen.

Hinweis – in Wirklichkeit kann da noch ein bisschen mehr passieren, da z.B. der Source-Path mehrere Verzeichnisse umfassen kann, oder der Compiler auch mit Class-Dateien und Jar-Files umgehen kann. Aber auch da passiert im Prinzip genau das gleiche wie hier beschrieben.

Der Source-Path hilft dem Compiler also referenzierte Klassen zu finden, um seinen Compile-Job mit möglichst vielen Überprüfung durchführen zu können.

4.4.2 Class-Path

Und was ist mit dem Class-Path? Das ist genau das gleiche für die virtuelle Maschine. Der Class-Path beschreibt, wo der Byte-Code der Klassen zu finden ist. D.h. referenziert er die Wurzel-Verzeichnisse für die Class-Dateien. Und daher müssen auch die Class-Dateien in korrespondierenden Verzeichnissen abgelegt werden, und haben auch den Namen der Klasse als Datei-Namen.

Wird kein Class-Path angeben, wird das aktuelle Verzeichnis als Class-Path angenommen. Folgende Aufrufe sind also identisch:

```
> java KlassenName
```

```
> java -cp . KlassenName
```

Der Class-Path kann aber auch über die Environment Variable „CLASSPATH“ einen Default bekommen, der dann statt des aktuellen Verzeichnisses genommen wird. In so einem Fall wären die beiden obigen Aufrufe nicht identisch, und der Class-Path müsste auch bei Class-Dateien im aktuellen Verzeichnis angegeben werden.

5 Datentypen und Variablen

5.1 Datentypen

5.1.1 Datentypen

Java ist eine statisch typisierte Sprache, d.h. jedes Objekt, Variable, Konstante, Literal, usw. hat einen eindeutigen Typ, der schon zur Compile-Zeit feststeht.

Java unterscheidet zwischen elementaren und benutzerdefinierten Datentypen.
- Elementare Datentypen sind fest in die Sprache eingebaut.
- Benutzerdefinierte Datentypen sind Klassen, Enums bzw. Interfaces.

Java unterscheidet zwischen statischen und dynamischen Typen:

- Der statische Typ ist der Typ, den der Compiler sieht.
- Der dynamische Typ ist der wahre Typ des Objekts zur Laufzeit.

Einen Unterschied zwischen statischen und dynamischen Typ kann es bei Objekten im Kontext von Vererbung geben.

5.2 Elementare Datentypen

Folgende elementaren Datentypen sind in Java enthalten:

Name	Grösse / Byte	Default	Minimum	Maximum
void	-	-	-	-
char	2	\u0000	\u0000	\uFFFF
boolean	1 Bit	false	-	-
byte	1	0	-128	127
short	2	0	-32768	32767
int	4	0	-2147483648	2147483647
long	8	0	-9223372036854775808	9223372036854775807
float	4	0.0	± 1.40239846 E - 45	± 3.40282347 E + 38
double	8	0.0	± 4.94065645841246544 E - 324	± 1.79769313486231570 E + 308

5.2.1 Typ-Umwandlungen

In Java können sogenannte Typ-Umwandlungen (auch Casts bzw. Type-Casts genannt) vorgenommen werden. Dies ist notwendig wenn ein Quelltyp nicht zum Zieltyp passt, z.B. bei einer Zuweisung oder einem Funktionsaufruf.

Eine Typ-Umwandlung wird durch die Angabe des Zieltyps in Klammern vor dem Quellausdruck vorgenommen.

```
byte b = (byte) 'A';
```

Hinweise:

- Sie sollten sich bemühen, Code zu schreiben, der keine oder nur wenige Typ-Umwandlungen benötigt.
- In Java sind nur Typ-Umwandlungen erlaubt, die syntaktisch Sinn machen.
- Typ-Umwandlungen gibt es nur im Kontext der elementaren Datentypen und innerhalb von Vererbungs-Hierarchien.
- Typ-Umwandlungen zwischen elementaren Datentypen werden benötigt, wenn der Quellausdruck nicht ohne Datenverlust in den Ziel-Typ passt – dann müssen Sie diese Wandlung explizit anfordern.
- Typ-Umwandlungen innerhalb von Vererbungs-Hierarchien können zur Laufzeit schief gehen, sie werfen dann eine Exception.

5.2.2 void

void ist ein Dummy-Typ, um bei Funktionen anzuzeigen, dass sie nichts zurückgeben.

```
void fct() {      // Diese Funktion gibt nichts zurueck
}
```

5.2.3 char

Ein char enthält ein Zeichen im Unicode Zeichensatz – d.h. 2 Byte Größe.

- Zeichenkonstanten werden in einfache Hochkommata gesetzt.
- Als Zeichenkonstanten sind normalen Zeichen, Escape-Sequenzen (z. B. ‘\n‘ für einen Zeilenumbruch oder ‘\t’ für einen Tabulator) und Unicode Sequenzen (die Angabe muss hexadezimal erfolgen, z. B. '\u03a3') erlaubt.

```
char c1 = 'A';          // Zeichen A
char c2 = '\n';         // Zeilenumbruch
char c3 = '\u03a3';     // Griechisches Summenzeichen (dezimal 931)
```

Ein char kann ohne Typ-Cast einem int, long, float und double zugewiesen werden. Ein char kann mit explizitem Typ-Cast einem byte und short zugewiesen werden – denken Sie aber an den möglichen Datenverlust!

```
byte b = (byte) 'A';
long l = 'A';
```

Umgekehrt kann einem char nur mit Typ-Cast ein integraler- oder Fliesskomma-Wert zugewiesen werden – denken Sie aber an den möglichen Datenverlust!

```
char c1 = (char) 42;
char c2 = (char) 3.14;
```

Achtung – vielleicht haben Sie sich gewundert, dass ein char und ein short nicht zuweisungskompatibel sind, sondern ein Typ-Cast nötig ist, wo doch beide Typen ein Grösse von 2 Byte haben. Aber bedenken Sie, dass ein char den Wertebereich von ‘\u0000’ bis ‘\uFFFF’ umfasst, d.h. keine negativen Zahlen aufweist, während ein short den Wertebereich von ‘-32768’ bis ‘+32767’ umfasst.

5.2.4 boolean

Der Typ „boolean“ ist der Typ für Wahrheitswerte, daher wenn ein Zustand nur falsch („false“) oder richtig (true) sein kann. Wir benutzen ihn auch häufig als Kriterium in den Java Konstrollstrukturen – wie z.B. If-Anweisungen oder Schleifen.

Ein boolean Wert kann nur „true“ oder „false“ enthalten.

```
boolean flag = true;
boolean okay = false;
```

Ein boolean Wert kann keiner Variablen eines anderen Typs zugewiesen werden, auch nicht durch Typ-Cast’s.

```
int i1 = true;            // Compiler-Error
int i2 = (int)true;       // Compiler-Error
```

Hinweis – diese Idee der Zuweisung eines Boolean an einen Int, wie auch die Idee im folgenden Beispiel, mag Sie vielleicht etwas verwundern. Wenn ja, dann ist das gut – und dann vergessen Sie dies direkt wieder. Aber es gibt Programmier-Sprachen, bei denen so etwas möglich ist – und für all die, die solche Sprachen kennen: in Java geht dies nicht.

Auch umgekehrt kann kein Wert eines anderen Typs einer boolean Variable zugewiesen werden, oder als boolean Wert benutzt werden, auch nicht durch Typ-Cast's.

```
boolean b = (boolean)1;    // Compiler-Error
int i=0;
if (i) {                   // Compiler-Error
```

5.2.5 Integrale Typen

Als integrale Typen stehen byte, short, int und long zur Verfügung. Zahlen-Literale sind normalerweise immer int, können aber auch einem byte oder short zugewiesen werden, wenn sie den entsprechenden Zahlenbereich nicht überschreiten. long-Zahlen-Literale werden durch ein 'l' oder 'L' hinter der Zahl deklariert.

```
byte b = 13;
short s = 42;
int i = 1234;
long l = 123456789012345L;
```

In Richtung der größeren Typen kann problemlos eine Zuweisung statt finden. In Richtung der kleineren nur über einen expliziten Typ-Cast - denken sie aber an den möglichen Datenverlust!

```
int i = 17;
short s = (short)i;
long l = i;
```

Jeder integrale Typ kann problemlos jedem Fliesskomma Typ zugewiesen werden – sie können dabei aber Genauigkeit verlieren!

```
long l = 123456789012345L
float f = l;
double d = l;
```

5.2.6 Fliesskomma Typen

Als Fliesskomma Typen stehen float und double zur Verfügung. Fliesskomma-Literale unterscheiden sich von den integralen Zahlen-Literalen dadurch, dass sie einen Punkt enthalten. Defaultmässig ist ein Fliesskomma-Literal immer vom Typ double, durch ein nachgestelltes 'f' oder 'F' wird es zu einem float Literal.

```
float f = 3.14F;
double d = 3.14;
```

Ein float-Wert kann problemlos einem double zugewiesen werden, umgekehrt geht dies nur mit einem expliziten Typ-Cast - denken sie aber an den möglichen Datenverlust!

```
float f1 = 3.14F;
double d1 = 3.14;
```

```
float f2 = (float)d1;
double d2 = f1;
```

5.3 Variablen

Variablen-Definiton

Syntax:

```
[Modifizierer] <Typ> <Name> [ = <Initialisierer>] ;
```

Bsp:

```
public final int i;
private String name = "Albert Einstein";
double d = 3.1415;
String s;
```

Variablen können nur in Klassen und in Funktionen definiert werden.

- Variablen in Klassen sind entweder Felder (Attribute oder Klassen-Variablen).
- Variablen in Funktionen sind lokale Variablen.

Näheres dazu später.

Variablen werden einfach über ihren Namen angesprochen, und können z.B. ausgewertet, verglichen und neu gesetzt werden.

```
public class Beispiel {

  public static void main(String[] args) {
    String name = "Albert Einstein";
    System.out.println("Mein Name ist " + name);
  }

}
```

5.4 Wert- und Referenz-Semantik

Variablen unterliegen je nach Typ einer unterschiedlichen Semantik.

- Variablen eines elementaren Datentyps unterliegen einer sogenannten Wert-Semantik.
- Variablen eines beliebigen anderen Datentyps unterliegen der Referenz-Semantik.

5.4.1 Wert-Semantik

Eine Variable hat Wert-Semantik, wenn sie den zu speichernden Wert direkt enthält, und alle Aktionen auf der Variablen direkt auf dem Wert stattfinden.

Nehmen wir z.B. eine Int-Variable „v1“, die mit dem Wert „42“ initialisiert wird.

```
int v1 = 42;
```

Für diese Variable wird irgendwo im Speicher ein 4 Byte großer Bereich reserviert, der den Wert „42“ enthält, und der über den Namen „v1“ angesprochen werden kann.

Wird jetzt eine zweite Int-Variable „v2“ definiert, die mit der Variablen „v1“ initialisiert wird, so bekommt „v2“ eine Kopie des Wertes von „v1“. Jede Variable enthält ihren eigenen Wert.

```
int v1 = 42;
int v2 = v1;

System.out.println(v1);          // 42
System.out.println(v2);          // 42
```

Das jede Variable ihren eigenen Wert enthält wird dann deutlich, wenn wir den Wert einer Variablen ändern – die andere behält ihren alten Wert bei.

```
int v1 = 42;
int v2 = v1;

System.out.println(v1);          // 42
System.out.println(v2);          // 42

v1 = 23;

System.out.println(v1);          // 23 <- veraendert
System.out.println(v2);          // 42 <- nicht veraendert
```

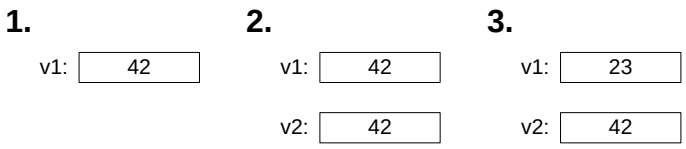

Abb. 5-1 : Wert-Semantik

Werte-Semantik liegt aber nicht nur bei einer Initialisierung oder einer Zuweisung vor, sondern z.B. auch bei Vergleichen und Funktions-Aufrufen.

Hier ein Beispiel mit zwei Vergleichen.

```
int v1 = 42;
int v2 = 42;

if (v1==v2) {
  System.out.println("v1 und v2 sind gleich");
}

if (v1==42) {
  System.out.println("v1 ist gleich 42");
}
```

Ausgabe

```
v1 und v2 sind gleich
v1 ist gleich 42
```

In beiden Fällen wird der Wert von „v1“ (d.h. die „42“) mit dem Wert des zweiten Operanden verglichen.

Diese Wert-Semantik gilt für alle Variablen eines elementaren Datentyps (d.h. char, boolean,

byte, short, int, long, float oder double), unabhängig davon ob die Variable eine lokale Variable, ein Parameter, eine Objekt- oder Klassen-Variable ist.

5.4.2 Referenz-Semantik

Abgesehen von den elementaren Datentypen werden alle Objekte (auch Arrays) in Java mit einer Referenz-Semantik angesprochen:

- Eine Variable auf einen nicht-elementaren Datentyp enthält nicht das Objekt selber, sondern nur eine Referenz auf das Objekt, man spricht auch von Referenz-Variablen.
- Eine Referenz-Variable wird mit null initialisiert.
- Vor der Nutzung einer Referenz-Variablen muss ihr ein Objekt zugewiesen werden.
- *‚Normale'* Referenz-Variablen-Zuweisungen (Operator =) kopieren das Objekt nicht, sondern erzeugen nur eine weitere Referenz auf das Objekt.
- Bei Funktionsaufrufen unterliegen natürlich auch die Parameter und der Rückgabewert von nicht-elementaren Datentypen der Referenz-Semantik. Z. B. können also die Original-Objekte von der aufgerufenen Funktion problemlos verändert werden.
- Der Zugriff auf Attribute und Funktionen von Objekten erfolgt über die Referenz-Variable und den Punkt-Operator.
- Der Vergleich zweier Referenz-Variablen vergleicht nicht die Werte der referenzierten Objekte (Wertgleichheit, bzw. tiefer Vergleich), sondern nur die Gleichheit des referenzierten Objekts (d.h. Identität, bzw. flacher Vergleich). Achtung – bei Strings kann es hier ein unerwartetes Verhalten geben.

```
StringBuilder buffer1 = null;                          // (1)
if (buffer1==null)
  System.out.println("buffer1 ist noch null");
else
  System.out.println("buffer1 ist: " + buffer1);

buffer1 = new StringBuilder("Hallo");                  // (2)
if (buffer1==null)
  System.out.println("buffer1 ist noch null");
else
  System.out.println("buffer1 ist: " + buffer1);

StringBuilder buffer2 = buffer1;                       // (3)
System.out.println("buffer2 ist: " + buffer2);

buffer1.append(" Welt");                               // (4)
System.out.println("buffer1 ist: " + buffer1);
System.out.println("buffer2 ist: " + buffer2);
```

Ausgabe

```
buffer1 ist noch null
buffer1 ist: Hallo
buffer2 ist: Hallo
buffer1 ist: Hallo Welt
buffer2 ist: Hallo Welt
```

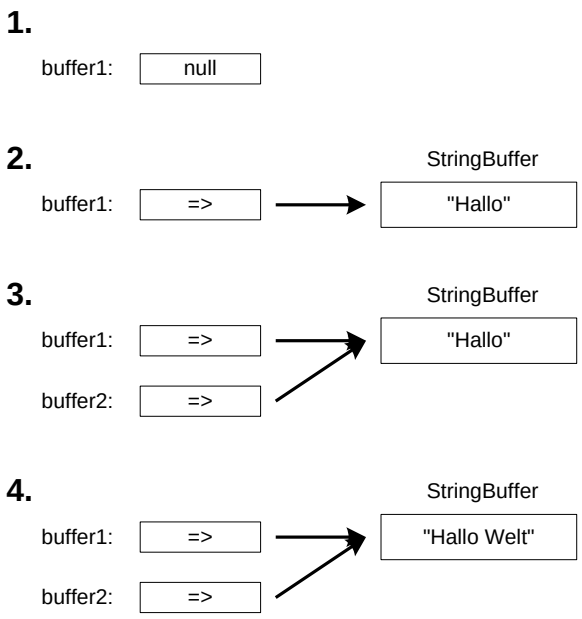

Abb. 5-2 : Referenz-Semantik

Achtung – auch Strings sind Objekte und daher führt der Vergleichs-Operator „==“ nur einen Identitätsvergleich durch, und keinen Wert-Vergleich. Dies ist ein gern gemachter Fehler, der um so kritischer ist, da der Code manchmal sogar funktioniert.

Wird versucht über eine Referenz-Variable mit dem Wert „null“ auf das referenzierte Objekt zuzugreifen (das ja nicht da ist, da ja nichts (null) referenziert wird), so wird dieser Fehler java-typisch mit einer Exception („java.lang.NullPointerException“) gemeldet.

```
String s = null;
s.length();         // Wirft NullPointerException
```