

# 6. Übersetzungseinheiten und Pakete

- Themen des Kapitels
  - Ausführbare Java-Programme
    - Übersetzungseinheiten und Gesamtprogramm
  - Laden von Klassen
    - Der Speicherbereich „Method Area“
  - Pakete (Packages)
    - Standard-Pakete
    - Erstellung eigener Pakete
    - Umgebungsvariable Classpath

# Ausführbare Java-Programme

- **Java-Programme** bestehen **nur aus Klassen und Interfaces**. Auch ausführbare Programme werden in Java mithilfe von Klassen definiert.
- Eine Klasse kann als Programm gestartet werden, wenn sie eine spezielle Klassenmethode namens **main** mit der Signatur

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

enthält.

- Übersetzungseinheit
  - Eine zu kompilierende Java-Datei ist eine **Textdatei** mit dem Suffix **.java**. Diese wird auch als **Übersetzungseinheit** bezeichnet.
  - Sie kann den Quelltext **einer** oder **mehrerer** Java-Klassen enthalten.
- Bei erfolgreicher Übersetzung mit einem **Java-Compiler** entstehen in dem Verzeichnis, das die Übersetzungseinheit enthält, Dateien mit dem Suffix **.class**, und zwar eine pro übersetzter Klasse.

# Ausführbare Java-Programme

- **Java-Programme** bestehen **nur aus Klassen und Interfaces**. Auch ausführbare Programme werden in Java mit **von Klassen** definiert.
- Eine Klasse kann als Programm gestartet werden, indem eine spezielle Klassenmethode namens **main** mit der

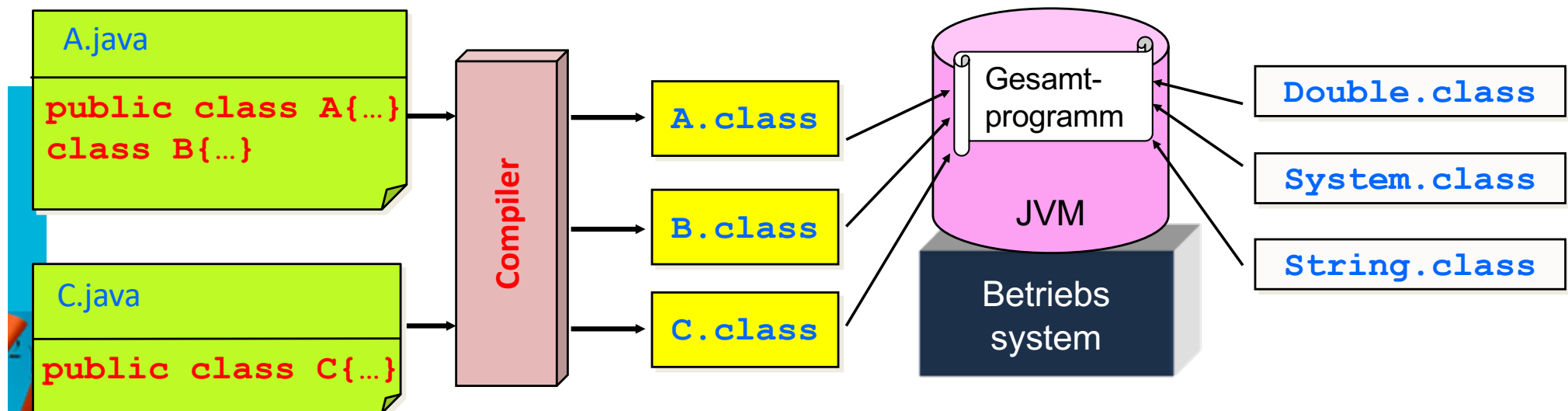
Lernen wir nächste Woche kennen.

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

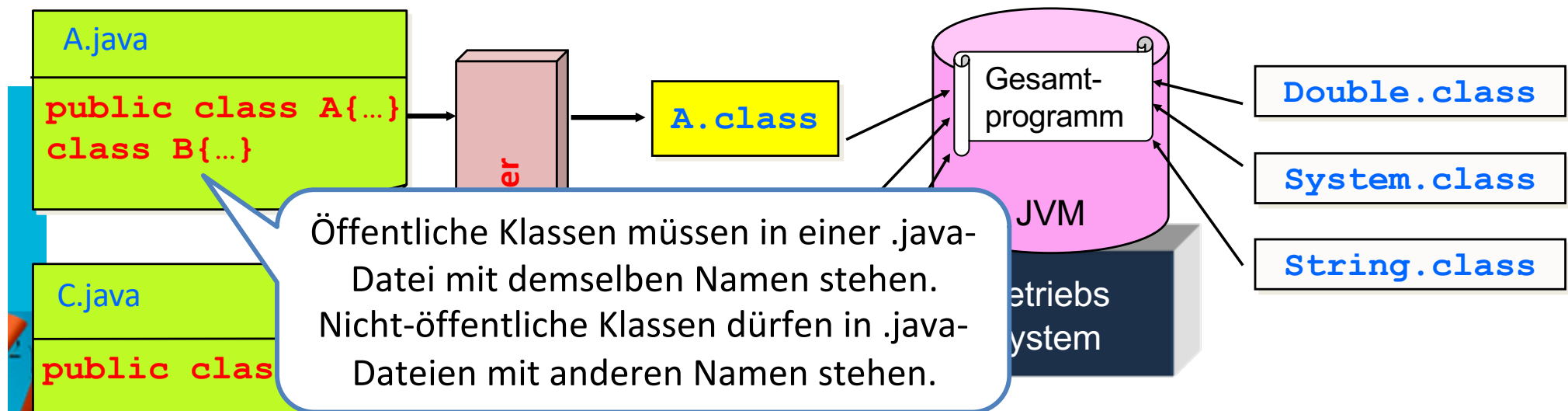
enthält.

- Übersetzungseinheit
  - Eine zu kompilierende Java-Datei ist eine **Textdatei** mit dem Suffix **.java**. Diese wird auch als **Übersetzungseinheit** bezeichnet.
  - Sie kann den Quelltext **einer** oder **mehrerer** Java-Klassen enthalten.
- Bei erfolgreicher Übersetzung mit einem **Java-Compiler** entstehen in dem Verzeichnis, das die Übersetzungseinheit enthält, Dateien mit dem Suffix **.class**, und zwar eine pro übersetzter Klasse.

# Übersetzung und Ausführung von Java-Programmen



# Übersetzung und Ausführung von Java-Programmen



# Entstehung des Gesamtprogramms

- Die Ausführung von Java-Programmen beginnt mit einer Klasse, die eine wie oben spezifizierte Methode **main** enthält.
  - Diese wollen wir **Hauptprogramm-Klasse** nennen.
- Die Hauptprogramm-Klasse kann andere Klassen benutzen, jene wieder andere Klassen etc.. Dies passiert erstmals durch:
  - Aufruf von statischen Methoden, Verwendung von statischen Datenfeldern und Aufruf von Konstruktoren
    - z. B. System.out
- Das **Gesamtprogramm** besteht aus der **Hauptprogramm-Klasse** und allen direkt oder indirekt **benutzten Klassen**.
  - Wenn eine Klasse in einem Programm zum **ersten Mal angesprochen** wird, wird sie zum Gesamtprogramm hinzugefügt. Man sagt dann, dass die Klasse **geladen** wird.
  - Der Zeitpunkt des Ladens kann auch tatsächlich etwas früher sein, aber das spielt aber für unsere Vorlesung keine Rolle.

# Initialisierung von Klassen

- Beim Laden einer Klasse wird ein **Speicherbereich für die statischen Felder** angelegt (und eine **Initialisierung der Klasse** vorgenommen).
  - Es werden noch weitere Informationen zur Klasse gespeichert.
- Zusätzlich zu dem Stack-Speicher und dem Heap-Speicher gibt es einen eigenen Speicherbereich (**Method Area**) für die Daten, die beim Laden einer Klasse angelegt werden.

Method Area

Speicherbereich für  
statische Felder

Speicherbereich für  
Bytecode

Stack-Speicher

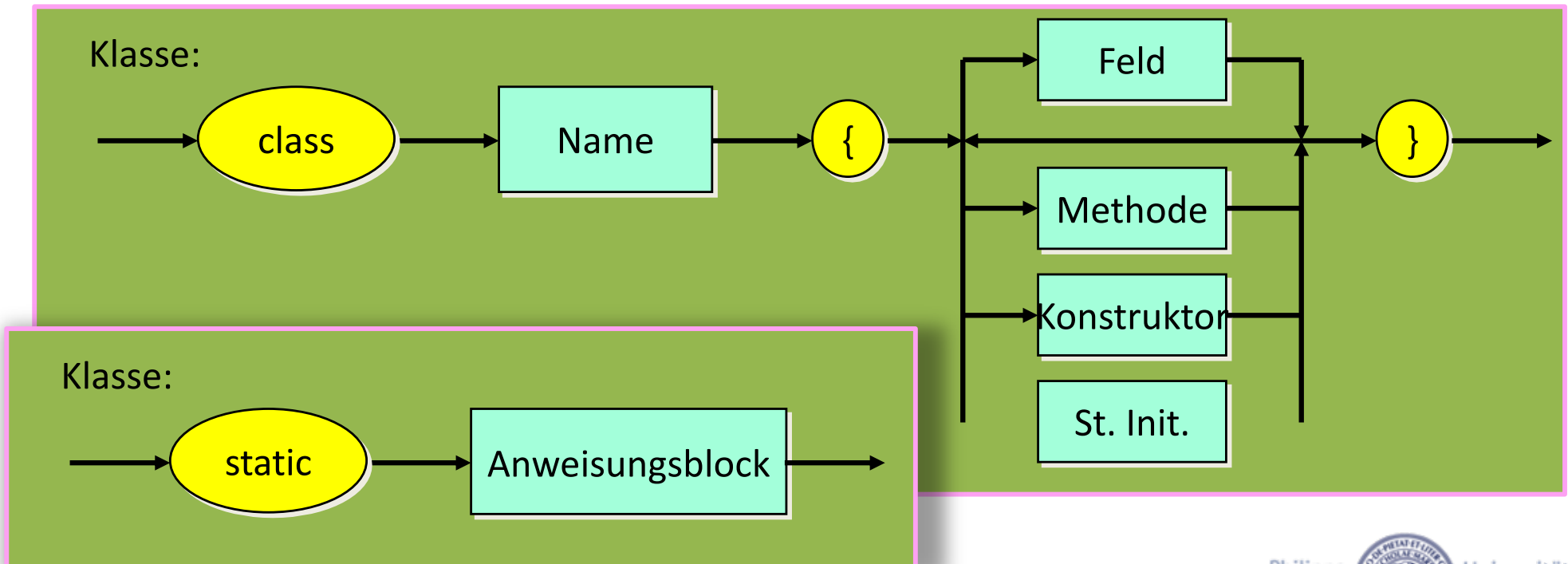
Speicherbereich für lokale Variablen  
und Parametervariablen

Heap-Speicher

Speicherbereich für Objekte

# Initialisierung von Klassen

- Übrigens: der "Static Initializer" einer Klasse ist so etwas wie eine Methode, die ausgeführt wird, wenn eine Klasse geladen wird.
  - Spezielle Syntax

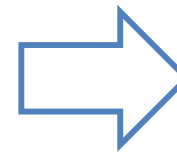




# Initialisierung von Klassen

- Static Initializer

```
class Main {  
    static {  
        System.out.println("Laden von Klasse Main");  
    }  
    public static void main(String[] args) {  
        System.out.println("In Klasse Main");  
        A a = new A();  
        a.m();  
    }  
}  
class A {  
    static {  
        System.out.println("Laden von Klasse A");  
    }  
    public A() {  
        System.out.println("Konstruktor von Klasse A");  
    }  
    public static void m() {  
        System.out.println("Methode m von Klasse A");  
    }  
}
```



Laden von Klasse Main  
In Klasse Main  
Laden von Klasse A  
Konstruktor von Klasse A  
Methode m von Klasse A

# Klassen, Klassen, Klassen, ...

- Durch die vielen Klassen (und Interfaces), die man typischerweise für das Gesamtprogramm erstellt, verliert man leicht den Überblick.
  - Wo finde ich passende Methoden?
  - Welche Klassen (und Interfaces) passen inhaltlich zueinander?



- Für die Erstellung großer Programme ist deshalb eine bessere Strukturierung hilfreich.

# Klassen, Klassen, Klassen, ...

- Klassenname: sollte das Konzept beschreiben, das die Klasse implementiert
- Was passiert, wenn zwei unterschiedliche Klassen durch denselben Namen beschrieben werden?
  - Zum Beispiel, weil Sie verschiedene Aspekte des Konzepts umsetzen
    - „Spieler“ in einem Online-Spiel: 1. als Akteur, 2. Speicherung der Nutzerdaten?
  - Zum Beispiel, weil der Name mehrere Bedeutungen hat
    - „Feder“ als Produkt in einem Shop werden: 1. Schreibfeder, 2. Druckfeder
  - Zum Beispiel, weil eine wiederverwendete Implementierung einen generischen Namen wie „Setup“ verwendet



# Pakete - Packages

- **Pakete** (engl.: packages) sind **Zusammenfassungen** von Java-Klassen für einen bestimmten Zweck oder einen bestimmten Typ von Anwendungen.
  - Dies hilft uns unsere eigenen Klassen besser zu strukturieren.
  - Funktionalität aus anderen Klassen leichter zu finden.
- Pakete sind **Namensräume**!
  - Klassennamen müssen innerhalb eines Pakets eindeutig sein, dürfen sich aber in anderen Paketen wiederholen
  - Der „voll qualifizierte“ (vollständige) Klassenname wird folgendermaßen gebildet:  
Paketname.Klassenname
- Zum Beispiel
  - logik.Spieler und daten.Spieler
  - schreibwaren.feder und eisenwaren.feder
  - meinprogramm.Setup und wiederverwendet.Setup

# Pakete - Packages

- Java-Programme können vordefinierte **Standard-Pakete** benutzen, die auf jedem Rechner mit einer Java-Ausführungsumgebung zu finden sein müssen.
- Das wichtigste Paket, **java.lang** enthält alle Klassen, die zum **Kern** der Sprache Java gezählt werden.
- Daneben enthält das JDK eine Unmenge weiterer Pakete, wie z.B.
  - **javax.swing** mit Klassen zur Programmierung von Fenstern und Bedienelementen,
  - **java.net** mit Klassen zur Netzwerkprogrammierung,
  - **java.util** mit nützlichen Klassen wie **Calendar**, **Date**, **ArrayList** und Schnittstellen **Collection**, **List**, **Iterator**, etc.

# Das Paket java.lang

- Kern dieser Standard-Pakete ist **java.lang** (lang steht für language.)
  - Dieses Paket enthält die wichtigsten vordefinierten Klassen, wie z. B. **System**, **String**.
  - Zusätzlich gibt es die „Wrapper-Klassen“ für die primitiven Datentypen: **Boolean**, **Character**, **Integer**, **Short**, **Byte**, **Long**, **Float** und **Double**.
  - Die Klasse **Math** enthält mathematische Standardfunktionen wie z.B. **sin**, **cos**, **log**, **abs**, **min**, **max**, etc. und mathematische Konstanten **E** und **PI**.
- Die Klassen aus **java.lang** können direkt verwendet werden.
- Für alle anderen Standard-Pakete muss man entweder **import-Anweisungen** hinzufügen, oder man muss alle Namen (inkl. des Paketnamens) voll ausschreiben.

# Import von Standardpaketen

- Um eine Klasse eines Paketes anzusprechen kann man grundsätzlich die Paketadresse dem Namen voranstellen, etwa

```
double[] arr = java.util.Arrays.copyOf(meinArray, 4711);  
java.util.Random r = new java.util.Random();
```

- Einfacher ist es, wenn man eine sogenannte **import-Anweisung** benutzt.
- Damit können eine oder alle Klassen eines Paketes **direkt angesprochen** werden, ohne ihnen jeweils die Paket-Adresse voranzustellen.

- Schickt man also eine Import-Anweisung wie

```
import java.util.Random;
```

voraus, so kann man anschließend Objekte und Methoden der Klassen Calendar benutzen, ohne den vollen Pfad zu ihnen anzugeben:

```
Random r = new Random();
```

- Die Namen von Standardpakete beginnen grundsätzlich mit
  - java**, oder **javax**
- Die zugehörigen Dateien liegen im Java-System.
  - Compiler und Interpreter wissen wie man diese Dateien findet.

# Einfach, aber nicht empfehlenswert

- Um alle Klassen eines Pakets zu importieren, kann man das Zeichen „\*“ als **Wildcard** einsetzen. So werden z.B. mit

```
import java.util.*;
```

sämtliche Klassen des Pakets **util** importiert oder genauer gesagt, werden die im Programm angesprochene Klassen **lediglich an den bezeichneten Stellen aufgesucht**.

- Nachteil dieser Variante
  - Man erkennt nicht mehr ohne weiteres die Abhängigkeiten seiner Klasse mit anderen Klassen, was aber bei der Fehlersuche in großen Programmen sehr nützlich ist.
- Seit Java 1.5 gibt es als zusätzliche Möglichkeit die **import-static-Anweisung**. Diese ermöglicht den unqualifizierten Zugriff auf statische Felder und Methoden einer Klasse.
- Man sollte diese Möglichkeit aber auch **mit Vorsicht** einsetzen. Ein kurzes Beispiel:

```
import static java.lang.Math.*;
import static java.lang.System.*;
class TestImportStatic {
    public static void main(String[] args) {
        out.println(sqrt(PI + E));
    }
}
```

2.420717761749361



# Einfach, aber nicht empfehlenswert

- Um alle Klassen eines Pakets zu importieren, kann man das Zeichen „\*“ als **Wildcard** einsetzen. So werden z.B. mit

```
import java.util.*;
```

sämtli  
Progra

So werden alle Klassen aus java.util in den aktuellen Namensraum importiert. Es kann wieder zu Namenskonflikten kommen.

auer gesagt, werden die im **bezeichneten Stellen** aufgesucht.

- Nacht

- Man
- was aber bei der Fehler
- in den Programmen sehr nützlich ist.



- Seit Ja
- ermög
- Klasse

Mit import-static kann es sogar zu Namenskonflikten mit lokalen Definitionen kommen. Z.B. könnte ein Feld „out“ in TestImportStatic nicht mehr unterschieden werden.

**ic-Anweisung**. Diese d Methoden einer

- Man se
- n. Ein kurzes Beispiel:

```
import static java.lang.Math.*;
import static java.lang.System.*;
class TestImportStatic {
    public static void main(String[] args) {
        out.println(sqrt(PI + E));
    }
}
```

2.420717761749361



# Beispiel: Scanner-Klasse aus java.util

- Seit dem JDK 1.5 wird die Klasse **Scanner** angeboten, mit deren Methoden man auf einfache Weise Daten einfacher Typen vom Konsolenfenster einlesen kann.
  - Methoden: `String next()`, `int nextInt()`, `double nextDouble()`, ....
- Dieses kleine Testprogramm kann von der Kommandozeile gestartet werden.

```
import java.util.Scanner;
public class TestScanner {
    public static void main (String[] args){
        Scanner s = new Scanner(System.in);
        System.out.print("Dein Vorname bitte: ");
        String name = s.next();
        System.out.println("Hallo " + name + "\nWie alt bist Du ?");
        int value = s.nextInt();
        if(value > 30)
            System.out.println("Hallo alter Hase");
        else
            System.out.println("Noch lange bis zur Rente ...");
        s.close();
    }
}
```

# Erstellung eigener Pakete

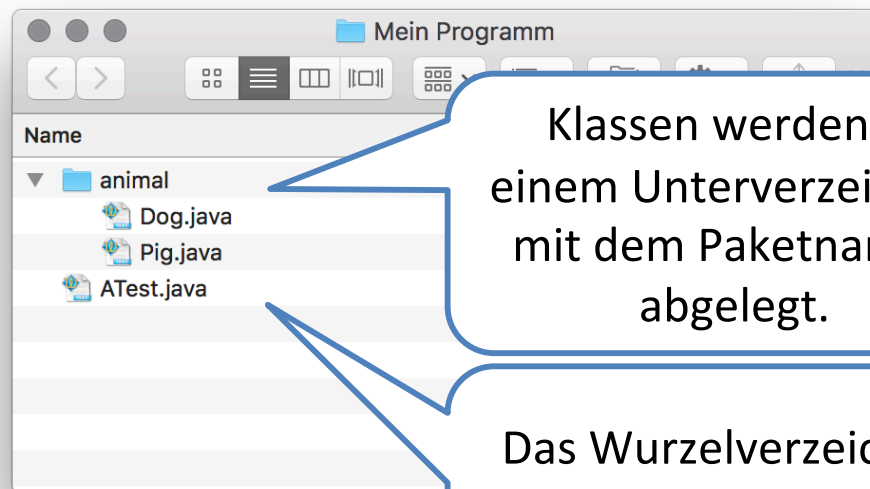
- Um selber Pakete zu erzeugen, kann man veranlassen, dass die Klassen einer Java-Datei zu einem bestimmten Paket gehören sollen.
- Dies erreicht man mit der **package-Anweisung** am Anfang der Übersetzungseinheit:

- **package meinPaketName;**

```
package animal;  
public class Pig{ ... }
```

```
package animal;  
public class Dog{ ... }
```

...



Klassen werden in einem Unterverzeichnis mit dem Paketnamen abgelegt.

Wurzelverzeichnis des Projekts nicht des Dateisystems.

Das Wurzelverzeichnis enthält das sogenannte „**Default Package**“.

Bei Klassen in diesem Package darf keine package-Anweisung angegeben sein.

# Verwendung eigener Pakete

- Sollen Klassen, Methoden und Felder aus anderen Paketen benutzt werden, müssen diese als **public** deklariert sein.
- Es soll ein Testprogramm erstellt werden, das die Klasse **Dog** unseres Pakets **animal** nutzt.
  - Dieses Testprogramm kann in irgendeinem Verzeichnis sein. Dieses wird damit zum **Default-Package**.

```
import animal.Dog;
public class ATest {
    public static void main(String args[]) {
        Dog a = new Dog();
    }
}
```

- *Die Klassen des Pakets animal müssen in dem Unterverzeichnis animal liegen*

Durch Setzen von CLASSPATH kann diese starre Regel noch aufgeweicht werden (→ später).

# Verwendung eigener Pakete

```
import animal.Dog;  
public class ATest {  
    public static void main(String args[]) {  
        Dog a = new Dog();  
    }  
}
```

Mein Programm — -bash — 70x12

```
[Christophs-MBP:Mein Programm bockisch$ javac ATest.java animal/*.java]  
Christophs-MBP:Mein Programm bockisch$
```

Mein Programm

Name	Size
▼ animal	--
Dog.class	187 bytes
Dog.java	989 bytes
Pig.class	187 bytes
Pig.java	734 bytes
ATest.class	288 bytes
ATest.java	122 bytes



# Namenskonventionen

- Wir hatten bisher einen **einfachen Namen** für das Package gewählt.
  - Man kann hier aber auch mit Paket- und Verzeichnishierarchien arbeiten.
- Wenn die Klasse **K** in dem Package **a.b.c** sein soll, müssen wir folgende Importanweisung verwenden: 

```
import a.b.c.K;
```
- Im Dateiverzeichnis muss **a** dann ein Unterverzeichnis des aktuellen Verzeichnisses (Default Packages) sein, **b** ein Unterverzeichnis von **a**, **c** ein Unterverzeichnis von **b** und **K.class** muss in **c** enthalten sein.
- Die Namen von Packages sollten eindeutig sein, weshalb es gewisse Konventionen gibt, die aber hier in der Vorlesung keine Rolle spielen.
  - Um Pakete von Klassen zu unterscheiden, sollten wir den Paktnamen mit einem kleinen Buchstaben und Klassennamen mit einem großen Buchstaben beginnen.

# Speicherorte der Pakete

- **Pakete** können in irgendeinem Verzeichnis auf dem eigenen Rechner liegen.
- In **frühen** Versionen von Java wurde propagiert, sie könnten auch auf einem Rechner **irgendwo im Internet** sein. Davon ist man wohl abgekommen:
  - aus Sicherheitsgründen,
  - aus Effizienzgründen und
  - um die Konsistenz von Versionen besser kontrollieren zu können.

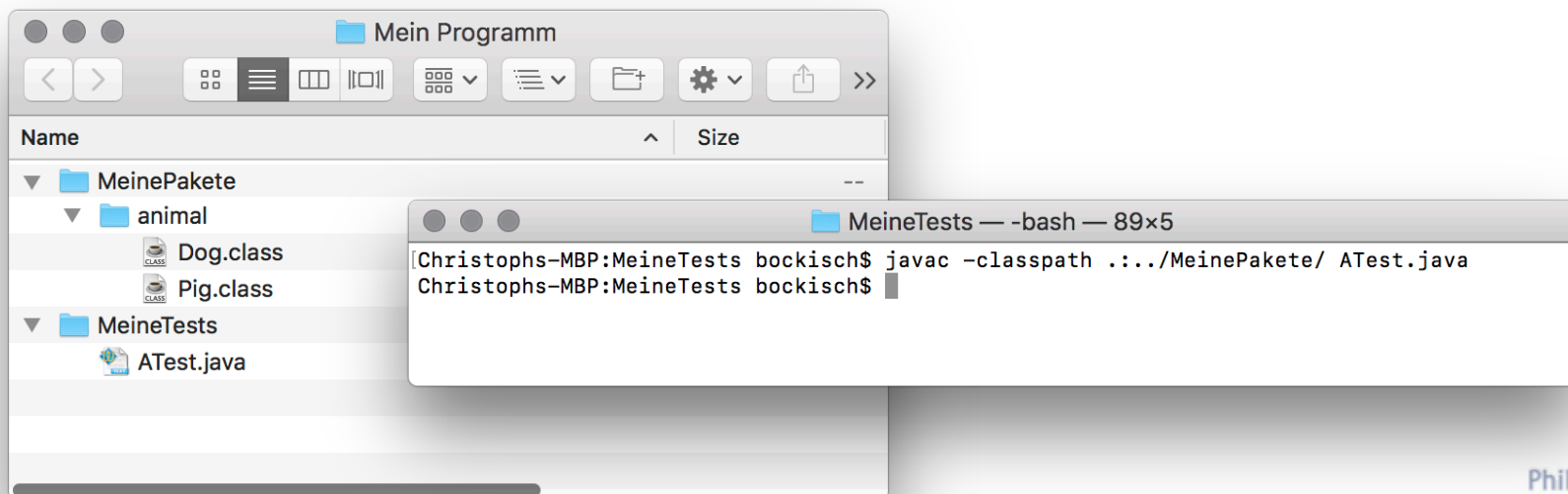
# Die Umgebungsvariable CLASSPATH

## Problem

- Bisher haben wir immer nur den Fall betrachtet, dass die Pakete in Unterverzeichnissen des aktuellen Verzeichnisses (Default Package) zu finden sind.
  - Das ist natürlich wenig flexibel und selten der Fall, wenn vorgefertigte Pakete benutzt werden sollen.

## Lösung

- Dem Compiler kann über die Option **-classpath** die Position anderer Pakete mitgeteilt werden





# Die Umgebungsv

## Problem

- Bisher haben wir immer nur den Fall betrachtet, dass die Pakete Unterverzeichnis sind.
  - Das ist natürlich benutzt werden

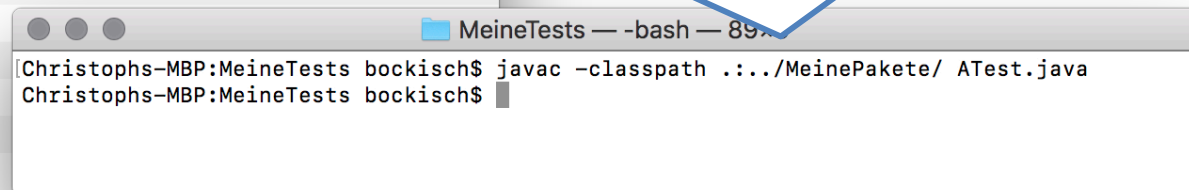
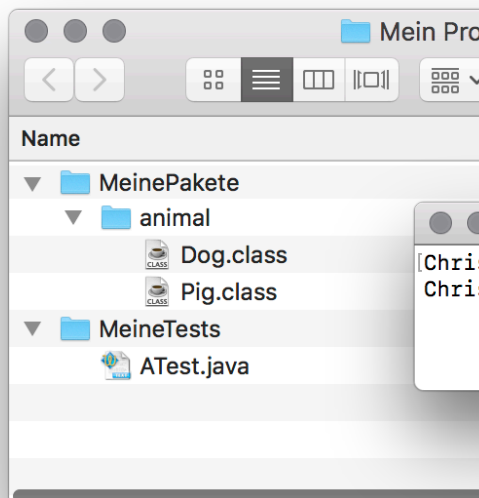
## Lösung

- Dem Compiler kann über die Option **-classpath** die Position anderer Pakete mitgeteilt werden

Unter Windows müsste in dem Beispiel der Classpath so angegeben werden:  
.;..\MeinePakete

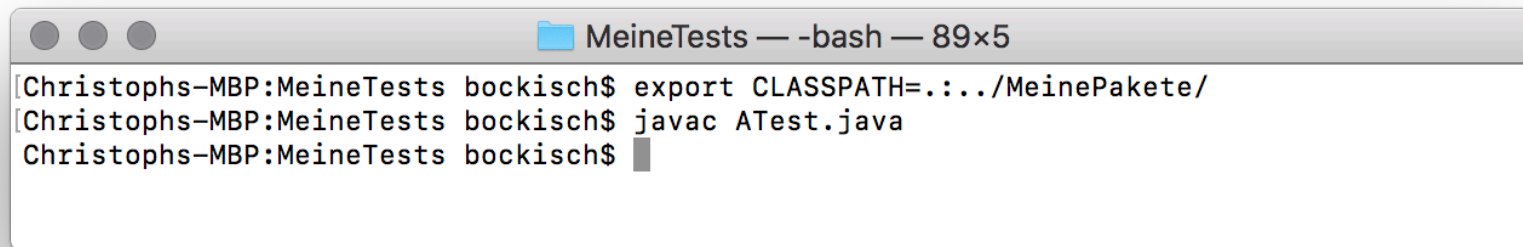
Achtung: Der Pfadseparator ist plattformspezifisch.  
Für Linux und macOS Doppelpunkt (:).  
Für Windows Semikolon (;)  
Auch Pfadangaben selbst sind plattformspezifisch.

Es wird eine Liste von Pfaden angegeben, die durchsucht werden. Die Pfade sind durch den „Pfad-Separator“ getrennt.



# Die Umgebungsvariable CLASSPATH

- Wenn häufig derselbe Classpath benötigt wird, kann auch die Umgebungsvariable **CLASSPATH** gesetzt werden.

A screenshot of a terminal window titled "MeineTests — -bash — 89x5". The terminal shows three lines of commands and their prompts: "[Christophs-MBP:MeineTests bockisch\$ export CLASSPATH=../../MeinePakete/]", "[Christophs-MBP:MeineTests bockisch\$ javac ATest.java]", and "Christophs-MBP:MeineTests bockisch\$".

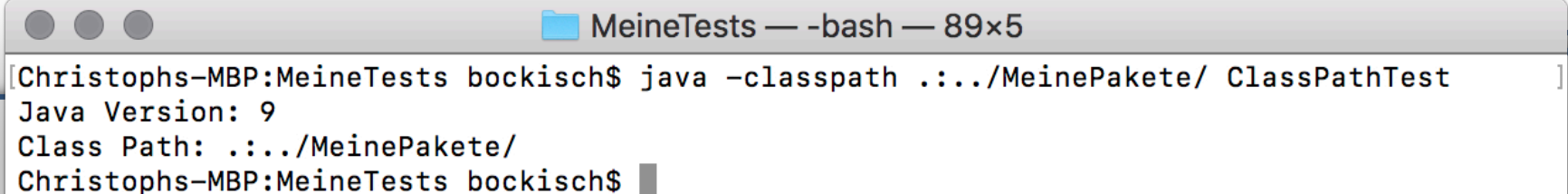
```
MeineTests — -bash — 89x5
[Christophs-MBP:MeineTests bockisch$ export CLASSPATH=../../MeinePakete/
[Christophs-MBP:MeineTests bockisch$ javac ATest.java
Christophs-MBP:MeineTests bockisch$
```

- Achtung: die Umgebungsvariable wird gelöscht, wenn Sie die Konsole schließen
  - Unter Linux/macOS können Sie das entsprechende Kommando der Datei `.bash_profile` hinzufügen
  - Für Windows siehe: <https://www.java.com/de/download/help/path.xml>

# Inspektion des Classpath

- Den Wert des Classpath kann man mit der Methode `getProperty` der Klasse `System` bekommen. Ein Beispiel hierzu:

```
class ClassPathTest {  
  
    public static void main(String[] args) {  
        System.out.println("Java Version: " + System.getProperty("java.version"));  
        System.out.println("Class Path: " + System.getProperty("java.class.path"));  
    }  
}
```



A terminal window titled "MeineTests — -bash — 89x5" shows the execution of the Java program. The command is `java -classpath ../../MeinePakete/ ClassPathTest`. The output is:  
Java Version: 9  
Class Path: ../../MeinePakete/  
The prompt returns to `Christophs-MBP:MeineTests bockisch$`.

- In diesem Fall enthält der **CLASSPATH** zwei Verzeichnisse:
  - Das **aktuelle Verzeichnis**, gekennzeichnet durch den Punkt (vor dem Pfadseparator)
    - Achtung: Wenn man den Punkt im CLASSPATH weglässt, bekommt man Probleme ....
  - **../../MeinePakete/**
  - Neben normalen Verzeichnissen sind auch **zip-** und **jar-Verzeichnisse** erlaubt

# Zusammenfassung

- Erstellung eines Gesamtprogramms
  - Bedarfsorientiertes Laden und Initialisieren der Klassen
- Aufteilen von Klassen in Pakete
  - Nutzen von Klassen aus anderen Paketen
  - Erstellen neuer Pakete
- Umgebungsvariable CLASSPATH