

Kapitel 03a: Systematische Abrundung bisheriges Java: Grundlegendes

Karsten Weihe

Fehler im Programm

Stichworte im Web: compile time error / run time error

Wir beginnen mit einer ersten kleinen systematischen Aufarbeitung des wichtigen Themas Fehler.

***Vorgriff:* In Kapitel 05 werden wir insbesondere Laufzeitfehler und deren Behandlung genauer betrachten.**

Fehler im Programm



Kompilierzeitfehler (compile-time errors):

- Schlüsselwort falsch verwendet
 - Wert einer Konstanten überschrieben
 - Variable / Konstante nicht ihrem Typ gemäß verwendet, z.B.
 - Arithmetische Operatoren bei boolean
 - Nichtboolesche Ausdrücke als Fortsetzungsbedingungen
 - Nichtexistierende Attribute / Methoden bei Klassen
 - Klammerung falsch
 - Konstrukte (z.B. Schleifen) falsch gebildet
 - ...
- Programm wird nicht übersetzt
- Statt dessen Fehlermeldung(en) vom Compiler

Aus vielfältiger eigener Erfahrung wissen Sie, dass es verschiedene Arten von Fehlern gibt, die dafür sorgen, dass der Compiler eine Fehlermeldung ausgibt und die Übersetzung des Programms nicht durchführt. Auf dieser Folie sehen Sie die wahrscheinlich wichtigsten Fälle. Das sind alles Fehlerarten, die schon vom Compiler erkannt werden können. Natürlich ist diese Aufzählung nicht abschließend, es gibt noch mehr solcher Fehlerarten.

Fehler im Programm



Laufzeitfehler (run-time errors):

- Bei ganzzahligen Typen: Division durch 0
 - Zugriff auf Attribut oder Methode von null
 - Zugriff auf nichtexistenten Arrayindex
 - Operator new findet keinen geeigneten Speicherplatz mehr
 - Kein Platz mehr auf dem Call-Stack
 - ...
- Ausführung des Programms wird abgebrochen
- Fehlermeldung zur Laufzeit

Andere Fehlerarten können hingegen erst zur Laufzeit festgestellt werden. Das Laufzeitsystem, das Ihr Programm ausführt, prüft jeden Schritt der Ausführung auf mögliche Fehler wie die hier aufgelisteten. Wird ein solcher Fehler entdeckt, bricht das Laufzeitsystem die Ausführung ab und gibt eine Fehlermeldung aus. Auch *diese* Auflistung ist nicht abschließend.

Fehler im Programm



Laufzeitfehler (run-time errors):

- **Bei ganzzahligen Typen: Division durch 0**
 - Zugriff auf Attribut oder Methode von null
 - Zugriff auf nichtexistenten Arrayindex
 - Operator new findet keinen geeigneten Speicherplatz mehr
 - Kein Platz mehr auf dem Call-Stack
 - ...
- Ausführung des Programms wird abgebrochen
- Fehlermeldung zur Laufzeit

Erinnerung: In Kapitel 01b, Abschnitt „Allgemein: Primitive Datentypen“, hatten wir gesehen, dass es für gebrochenzahlige Datentypen tatsächlich eine Konstante namens NaN gibt, die das Ergebnis einer Division durch 0 repräsentiert.

Das ist bei ganzzahligen Datentypen nicht der Fall. Hier gibt es keine solche Repräsentation, und die Division durch 0 führt zu Programmabbruch und Fehlermeldung.

Fehler im Programm



Laufzeitfehler (run-time errors):

- Bei ganzzahligen Typen: Division durch 0
 - Zugriff auf Attribut oder Methode von null
 - Zugriff auf nichtexistenten Arrayindex
 - Operator new findet keinen geeigneten Speicherplatz mehr
 - Kein Platz mehr auf dem Call-Stack
 - ...
- Ausführung des Programms wird abgebrochen
- Fehlermeldung zur Laufzeit

***Erinnerung* von Kapitel 01d, Abschnitt zum Literal null: Eine Referenz muss nicht unbedingt auf ein Objekt verweisen, sondern kann statt dessen den symbolischen Wert null enthalten.**

Wenn Sie von irgendwoher eine Referenz bekommen – zum Beispiel als Parameter einer Methode, die Sie gerade schreiben –, und Sie wissen nicht sicher, dass diese Referenz tatsächlich auf ein Objekt verweist, dann sollten Sie besser auf `== null` testen, bevor Sie über diese Referenz lesend oder schreibend auf ein Attribut des vermeintlichen Objektes zugreifen oder eine Methode damit aufrufen. Denn wenn die Referenz den Wert null hat, bricht das Laufzeitsystem die Ausführung des Programms mit einer entsprechenden Fehlermeldung ab.

Fehler im Programm



Laufzeitfehler (run-time errors):

- Bei ganzzahligen Typen: Division durch 0
 - Zugriff auf Attribut oder Methode von null
 - Zugriff auf nichtexistenten Arrayindex
 - Operator new findet keinen geeigneten Speicherplatz mehr
 - Kein Platz mehr auf dem Call-Stack
 - ...
- Ausführung des Programms wird abgebrochen
- Fehlermeldung zur Laufzeit

Ein Array, das mit einem nichtnegativen ganzzahligen Wert n eingerichtet wird, hat Indexbereich 0 bis n minus 1. Lesender oder schreibender Zugriff auf einen negativen Index oder auf einen Index größer-gleich n führt ebenfalls zum Prozessabbruch mit Fehlermeldung.

Fehler im Programm



Laufzeitfehler (run-time errors):

- Bei ganzzahligen Typen: Division durch 0
 - Zugriff auf Attribut oder Methode von null
 - Zugriff auf nichtexistenten Arrayindex
 - Operator new findet keinen geeigneten Speicherplatz mehr
 - Kein Platz mehr auf dem Call-Stack
 - ...
- Ausführung des Programms wird abgebrochen
- Fehlermeldung zur Laufzeit

Vorgriff: In Abschnitt zum Garbage Collector in Kapitel 03b werden wir sehen, dass das eigentlich nicht passieren sollte, aber doch passieren kann, wenn sie nur oft genug neuen Speicherplatz mit Operator new anfordern und nicht dafür sorgen, dass das Laufzeitsystem alten Speicherplatz wieder freigeben kann.

Fehler im Programm



Laufzeitfehler (run-time errors):

- Bei ganzzahligen Typen: Division durch 0
 - Zugriff auf Attribut oder Methode von null
 - Zugriff auf nichtexistenten Arrayindex
 - Operator new findet keinen geeigneten Speicherplatz mehr
 - **Kein Platz mehr auf dem Call-Stack**
 - ...
- Ausführung des Programms wird abgebrochen
- Fehlermeldung zur Laufzeit

Erinnerung: Den Call_Stack hatten wir in Kapitel 01e, Abschnitt „Ausführung von Methoden“, gesehen.

Auch für den Call-Stack ist nur begrenzter Speicherplatz reserviert. Normalerweise reicht der aus. Wenn man das Programm aber so gestaltet, dass irgendwann so viele Frames gleichzeitig auf dem Call-Stack sind, dass dieser Speicherplatz vollständig aufgebraucht ist, und Sie dann noch einen weiteren Methodenaufruf starten, dann passt der zugehörige Frame nicht mehr auf den Call-Stack, und die Ausführung Ihres Programms wird abgebrochen.

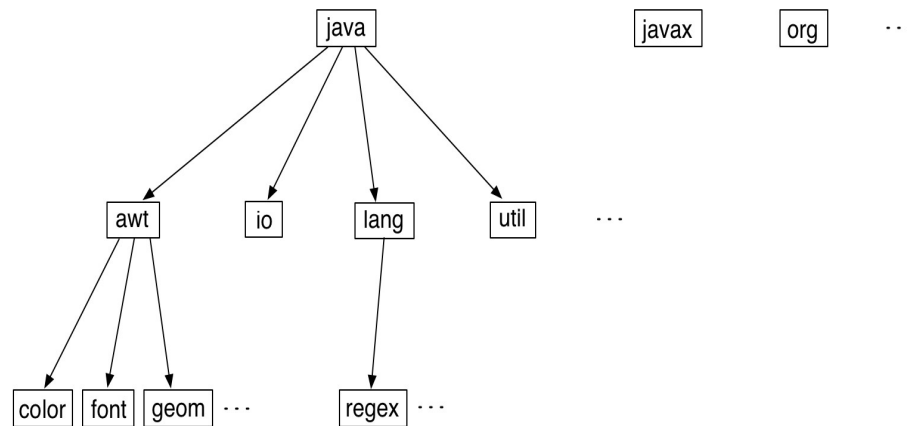
Vorgriff: In Kapitel 04a werden wir eine Situation sehen, in der diese Gefahr real ist, Stichwort Rekursion.

Packages und Standardbibliothek

Oracle Java Tutorials: Creating and Using Packages

Erinnerung: In jeder von uns bereitgestellten Quelltextdatei finden Sie das Schlüsselwort `import` für das Importieren von Funktionalität aus einem Package. In Kapitel 01f, Abschnitt „Zugriffsrechte und Packages“, haben Sie neben `import` auch schon das Schlüsselwort `package` gesehen, mit dem gesagt wird, dass die Funktionalität in dieser Quelltextdatei zu dem Package gehören soll, dessen Name hinter dem Schlüsselwort `package` steht.

Standardbibliothek

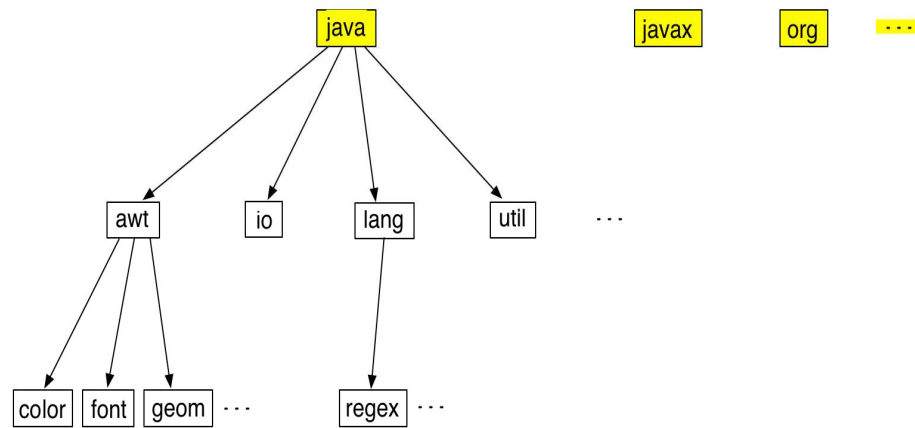


Packages sind hierarchisch organisiert, ähnlich wie Verzeichnisse von Dateien. Tatsächlich sind sie auch in Verzeichnissen und Unterverzeichnissen abgelegt, und die Verzeichnisstruktur ist identisch zur Package-Struktur.

Auf dieser Folie sehen Sie einen sehr kleinen Ausschnitt aus der hierarchischen Package-Struktur, in die die Java-Standardbibliothek zerlegt ist. Jede Distribution von Java mit Compiler und Laufzeitsystem enthält auch die Standardbibliothek, so dass Sie sich blindlings darauf verlassen können, dass die darin enthaltene Funktionalität überall vorhanden ist, wo Ihr Programm vielleicht einmal laufen könnte – und zwar überall exakt dieselben Definitionen von Klassen, Interfaces, Methoden und so weiter.

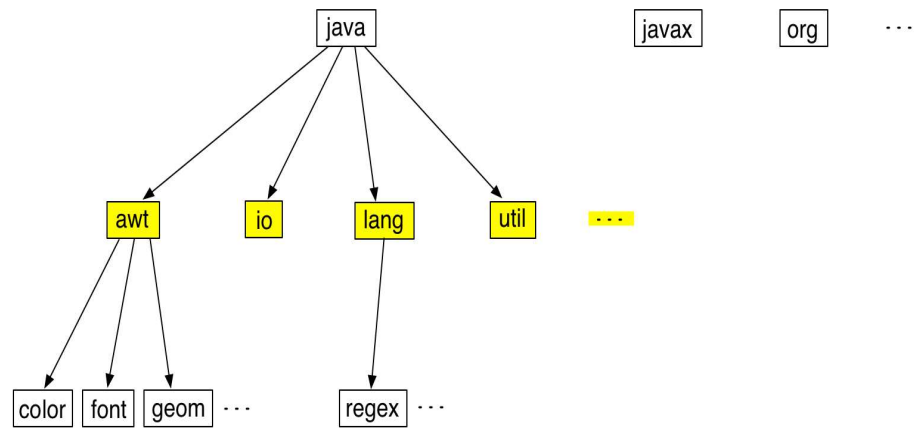
***Aber:* Im Laufe der Jahre und Jahrzehnte wurden etliche Versionen der Java-Standardbibliothek entwickelt. Sie müssen aufpassen, ob Ihre Version passt, und ggf. entweder Ihre Programme anpassen oder die Distribution upgraden. Das kennen Sie ja von eigentlich allen anderen IT-Tools genauso.**

Standardbibliothek



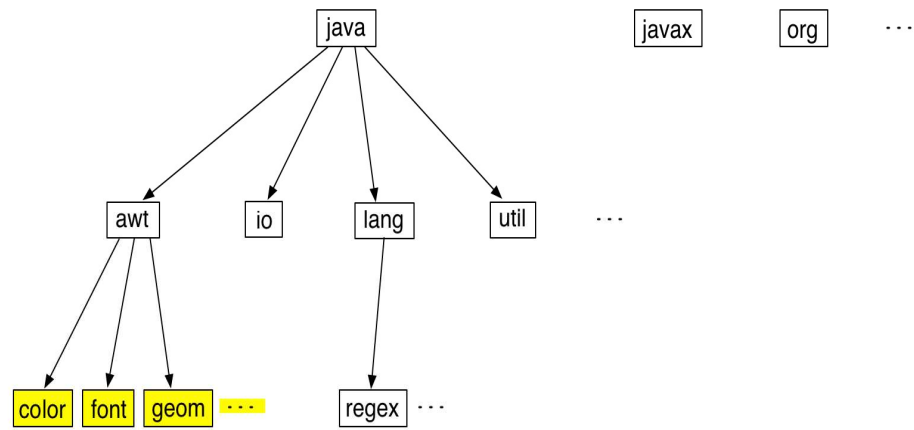
Die Standardbibliothek ist auf mehrere übergeordnete Packages verteilt.

Standardbibliothek



Jedes dieser übergeordneten Packages enthält mehrere Subpackages
...

Standardbibliothek



... und diese wiederum Subpackages und so weiter, aber nur punktuell noch tiefer als hier angedeutet.

Standardbibliothek



java	java.lang
java.awt	java.lang.regex
java.awt.color	java.util
java.awt.font	javax
java.awt.geom	org
java.io	

So lauten die Namen all dieser Packages und Subpackages aus dem Bild auf der letzten Folie. Sie sehen, dass die Reihenfolge der einzelnen Namensbestandteile von übergeordnet nach untergeordnet geht, wie es ja auch auf gängigen Betriebssystemen bei Verzeichnisnamen ist. Bei Package-Namen in Java ist der Punkt das Trennsymbol für die Namensbestandteile.

Voll qualifizierter Name:

java.lang.String

java.awt.Graphics

java.awt.color.Color

Der voll qualifizierte Name einer Klasse, eines Interface oder einer Enumeration besteht aus dem vollständigen Package-Namen und dem eigentlichen Namen, beides getrennt durch einen weiteren Punkt.

Importieren

```
import java.util.*;  
  
import java.awt.color.Color;  
  
import java.awt.color.*;
```

Wir haben schon mehrfach gesehen, dass man den in der Form wie in der ersten und der dritten Zeile den Inhalt eines Package vollständig importiert. In der zweiten Zeile wird nur die eine Klasse namens Color importiert, sonst nichts aus diesem Package.

Importieren

```
import java.awt.*;
```

```
Graphics graphic; // java.awt.Graphics
```

```
Color color; // java.awt.color.Color
```

Wenn Sie eine Klasse oder auch ein Interface oder eine Enumeration entweder allein oder mit dem ganzen Package importieren, dann können Sie diese Typen in Ihrer Quelltextdatei später einfach nur mit ihrem Namen ansprechen.

Importieren

```
import java.awt.*;
```

```
Graphics graphic; // java.awt.Graphics
```

```
Color color; // java.awt.color.Color
```

Die Inhalte von Subpackages werden aber nicht mitimportiert. Will man Inhalte aus dem Subpackage importieren, muss man dieses zusätzlich explizit importieren.

Importieren



```
import java.lang.*;
```

```
// Automatisch importiert
```

Dieses Package ist so wichtig, dass es immer automatisch vom Compiler importiert wird, Sie brauchen es nie selbst zu importieren. Die obere Zeile ist also überflüssig.

Eigene Packages



Am Anfang jeder Quelldatei, die zu *mypackage* gehören soll, die Zeile:

package mypackage;

- Muss *erste* Zeile sein
- Nur *eine* package-Zeile in jeder Quelldatei
- Dieses Package wird automatisch importiert

Auch das Hinzufügen einer Quelldatei zu einem Package haben wir schon gesehen.

Eigene Packages



Am Anfang jeder Quelldatei, die zu *mypackage* gehören soll, die Zeile:

```
package mypackage;
```

- Muss *erste* Zeile sein
- Nur *eine* package-Zeile in jeder Quelldatei
- Dieses Package wird automatisch importiert

Diese Zeile muss wirklich die allererste in der Quelltextdatei sein.

Eigene Packages



Am Anfang jeder Quelldatei, die zu *mypackage* gehören soll, die Zeile:

package mypackage;

- Muss *erste* Zeile sein
- Nur *eine* package-Zeile in jeder Quelldatei
- Dieses Package wird automatisch importiert

Da es keine zwei allerersten Zeilen geben kann, gilt dies schon automatisch. Es ist also jede Quelltextdatei mit allen ihren Definitionen entweder ganz oder gar nicht in einem Package drin, und zwar auch nur in maximal einem.

Eigene Packages



Am Anfang jeder Quelldatei, die zu *mypackage* gehören soll, die Zeile:

package mypackage;

- Muss *erste* Zeile sein
- Nur *eine* package-Zeile in jeder Quelldatei
- Dieses Package wird automatisch importiert

Die package-Zeile enthält implizit auch die import-Zeile für dasselbe Package. Alles, was noch so in diesem Package vorhanden ist, ist also automatisch importiert.

In jeder Quelldatei

- ***Eine* public-Klasse oder *ein* public-Interface oder *eine* public-Enumeration**
 - **Außerhalb des Packages sichtbar**
 - **Klassenname = Dateiname**
- **Beliebig viele Hilfsklassen / -interfaces / -enumerationen**
 - ***Nicht* außerhalb des Packages sichtbar**

Diesen Punkt haben wir auch schon gesehen: Eine einzelne Definition einer Klasse, eines Interface oder einer Enumeration darf **public** sein, und der Name dieser Entität ist dann der **Dateiname**, natürlich zuzüglich Dateiendung.

In jeder Quelldatei

// Datei B.java:

interface A { ... }

public class B { ... }

class C { ... }

Hier noch einmal ein kleines Beispiel zur Illustration.

In jeder Quelldatei

// Datei B.java:

interface A { ... }

public class B { ... }

class C { ... }

Hier ist die eine erlaubte public-Definition, die auch den Namen der Datei determiniert.

In jeder Quelldatei

// Datei B.java:

interface A { ... }

public class B { ... }

class C { ... }

Und hier sind beispielhaft noch zwei weitere Definitionen in derselben Datei, die dann beide *nicht* public sein dürfen.

Warum Packages?

- **Aufeinander bezogene komplexe Typen bilden eine Gruppe**
- **Klassen / Interfaces / Enumerations sind durch Gruppierung leichter auffindbar**
- **Privilegierter Zugriff innerhalb des Packages**
- **Namenskonflikte kontrollierbar**

Natürlich stellt sich die Frage, warum ein solcher Aufwand getrieben wird, dass Java-Programme in eine derartige zusätzliche hierarchische Struktur gepresst werden – übrigens bei so ziemlich jeder anderen gängigen Sprache ebenfalls, nur mit anderer Schreibweise.

Warum Packages?

- **Aufeinander bezogene komplexe Typen bilden eine Gruppe**
- **Klassen / Interfaces / Enumerations sind durch Gruppierung leichter auffindbar**
- **Privilegierter Zugriff innerhalb des Packages**
- **Namenskonflikte kontrollierbar**

Gruppierung ist das Zauberwort. Viele Klassen, Interfaces und Enumerations gehören inhaltlich zusammen und sollten daher zu einer Einheit zusammengefasst werden.

Warum Packages?

- **Aufeinander bezogene komplexe Typen bilden eine Gruppe**
- **Klassen / Interfaces / Enumerations sind durch Gruppierung leichter auffindbar**
- **Privilegierter Zugriff innerhalb des Packages**
- **Namenskonflikte kontrollierbar**

Ein wichtiger Vorteil ist, dass man nützliche Funktionalität durch inhaltliche Gruppierung leichter auffinden kann. Oft weiß man ja gar nicht die Namen, nach denen man suchen müsste, oder man weiß gar nicht, ob die Funktionalität, nach der man sucht, überhaupt vorhanden ist. Durch die Gruppierung nach Sachthemen kann man sich auf der Suche nach Funktionalität auf ein oder wenige Packages beschränken, in denen die Funktionalität gemäß thematischer Aufteilung zu finden sein müsste.

Warum Packages?

- **Aufeinander bezogene komplexe Typen bilden eine Gruppe**
- **Klassen / Interfaces / Enumerations sind durch Gruppierung leichter auffindbar**
- **Privilegierter Zugriff innerhalb des Packages**
- **Namenskonflikte kontrollierbar**

Wir haben schon gesehen und werden gleich noch einmal sehen, dass es möglich ist, den Klassen innerhalb desselben Packages Zugriffe zu erlauben, die Klassen außerhalb des Package nicht erlaubt sind. Mit Zugriff ist lesender und schreibender Zugriff auf Attribute sowie Aufruf von Methoden gemeint.

Wofür ist das gut: Wir haben ja schon gesehen, dass man den Zugriff einzelner Funktionalitäten einer Klasse durch Schlüsselwort `private` auf die Klasse selbst beschränken kann und dass das auch gut ist. Packages erlauben solche Zugriffsbeschränkungen eben nun auch auf der Ebene mehrerer inhaltlich aufeinander bezogener Klassen.

Warum Packages?

- **Aufeinander bezogene komplexe Typen bilden eine Gruppe**
- **Klassen / Interfaces / Enumerations sind durch Gruppierung leichter auffindbar**
- **Privilegierter Zugriff innerhalb des Packages**
- **Namenskonflikte kontrollierbar**

Der entscheidende Grund für die Einführung von Packages in eigentlich allen ernstzunehmenden Programmiersprachen ist aber der hier genannte. Dafür sehen wir uns auf den nächsten Folien ein illustratives Beispiel an.

Namenskonflikte

Beispiel:

▪ X und Y werden aus
mypackage1
exportiert

```
import mypackage1.*;  
import mypackage2.*;
```

.....

▪ X und Z werden aus
mypackage2
exportiert

```
Y y = new Y ();
```

```
Z z = new Z ();
```

```
X x = new X ();
```

Links ist die Ausgangssituation beschrieben, rechts ein kleines Codefragment auf Basis dieser Ausgangssituation.

Namenskonflikte

Beispiel:

- X und Y werden aus mypackage1 exportiert

- X und Z werden aus mypackage2 exportiert

```
import mypackage1.*;  
import mypackage2.*;
```

.....

```
Y y = new Y ();
```

```
Z z = new Z ();
```

```
X x = new X ();
```

Die beiden Typen Y und Z sind jeweils nur in einem der beiden Packages public, das heißt, außerhalb des jeweiligen Packages potentiell sichtbar und importierbar. Daher gibt es keinen Namenskonflikt.

Namenskonflikte

Beispiel:

- X und Y werden aus mypackage1 exportiert

- X und Z werden aus mypackage2 exportiert

```
import mypackage1.*;  
import mypackage2.*;
```

.....

```
Y y = new Y ();
```

```
Z z = new Z ();
```

```
X x = new X ();
```

Der Typ X ist hingegen in beiden Packages public, so haben wir das Beispiel konstruiert. Hier entsteht ein unauflösbarer Namenskonflikt, denn der Compiler kann ja nicht wissen, welches X gemeint ist – das aus mypackage1 oder das aus mypackage2.

Namenskonflikte



```
import mypackage1.*; // u.a. X wird exportiert  
import mypackage2.*; // u.a. X wird exportiert
```

```
mypackage1.X x1 = new mypackage1.X ();
```

```
mypackage2.X x2 = new mypackage2.X ();
```

Durch Qualifizierung des Namens X mit dem Namen des Package lässt sich dieser Namenskonflikt auflösen.

Zwei Typen mit demselben Namen:

- Im selben Package:

- Geht gar nicht
- Auch nicht bei nicht-public Typen

- In unterschiedlichen Packages:

- Mit Packagenamen vorneweg ok

Auf dieser Folie schauen wir uns das Thema Namenskonflikte noch einmal im Überblick an.

Zwei Typen mit demselben Namen:

- **Im selben Package:**

- **Geht gar nicht**

- **Auch nicht bei nicht-public Typen**

- **In unterschiedlichen Packages:**

- **Mit Packagenamen vorneweg ok**

Die Sprache Java verbietet grundsätzlich die Vergabe desselben Namens für zwei oder mehr Typen in einem Package.

Zwei Typen mit demselben Namen:

- **Im selben Package:**

- **Geht gar nicht**

- **Auch nicht bei nicht-public Typen**

- **In unterschiedlichen Packages:**

- **Mit Packagenamen vorneweg ok**

Das gilt auch für die Typen, die *nicht* vom Package exportiert werden, denn Namenskonflikte innerhalb des Package haben ja mit public oder nicht public gar nichts zu tun.

Zwei Typen mit demselben Namen:

- Im selben Package:

- Geht gar nicht
- Auch nicht bei nicht-public Typen

- In unterschiedlichen Packages:

- Mit Packagenamen vorneweg ok

Wie wir eben gesehen haben, können wir Namenskonflikte auflösen, die aus dem Import desselben Namens aus zwei verschiedenen Packages entstehen.

Firmen und Institutionen:

- Der umgedrehte Domain-Name
- Unzulässige Zeichen → Underscore

de.tu_darmstadt.informatik.algo.fop

Inzwischen werden selbstgeschriebene Typen ja weltweit zur freien oder kommerziellen Mitbenutzung angeboten. Wenn Leute an unterschiedlichen Orten in der Welt sich unabhängig voneinander Gedanken machen, wie sie ihre Packages benennen sollen, werden verschiedene Packages denselben Namen bekommen und daher nicht zugleich importierbar sein. Dieses Problem ist durch weltweit einheitliche Namenskonventionen gelöst.

Firmen und Institutionen:

- Der umgedrehte Domain-Name
- Unzulässige Zeichen → Underscore

de.tu_darmstadt.informatik.algo.fop

Zunächst einmal ist es generelle Konvention, dass Packagenamen nur aus Kleinbuchstaben bestehen. Packagenamen sollen eher kurz sein, so dass sich das Problem mit den Wortanfängen innerhalb eines Namens gar nicht stellt. Will man ein Zeichen ausdrücken, das für Identifier nicht erlaubt ist, muss man es eben anderweitig ausdrücken. In der Regel sind das Trennzeichen wie der Bindestrich, dann ist der Underscore ein zulässiger und auch einigermaßen guter Ersatz.

Firmen und Institutionen:

- Der umgedrehte Domain-Name
- Unzulässige Zeichen → Underscore

`de.tu_darmstadt.informatik.algo.fop`

Dieses Package existiert nicht, es ist rein illustrativ gewählt.

- **Seit Java 9 sind Packages in Modulen zusammengefasst**
 - **Bieten bessere Möglichkeiten als Packages zur Steuerung und Kontrolle der Abhängigkeiten in Software**
- **In der FOP aber nicht behandelt**
 - **Nur hier ganz kurz angesprochen**

Zum Abschluss dieses Abschnitts noch ein ganz kleiner Ausblick auf ein Thema, das in der FOP nicht thematisiert werden kann, das Sie sich später bei Bedarf aber auch gut selbst auf Basis der FOP erarbeiten können sollten.

Module



- **Fast alle von uns verwendete Funktionalität aus der Standardbibliothek sind in Modul `java.base`**
 - Wird automatisch in jedem Java-Programm importiert
 - Für uns in der FOP ändert sich überhaupt nichts
- **Ausnahme: Swing-Klassen auf Übungsblatt 10**
 - Sind in Modul `java.desktop`
 - Import wird auf dem Übungsblatt erklärt

Für Ihre Arbeit an den Übungsblättern, am Projekt und in der Klausur ändert sich eigentlich gar nichts; alle Packages können wie gewohnt einfach so importiert werden, ohne dass Sie etwas zu Modulen wissen müssen. Die eine Ausnahme ist auf dieser Folie benannt.

Lokale Variable / Konstante

vs.

Attribute

vs.

Parameter

Bevor wir zum Scope von Identifiern kommen, brauchen wir noch eine kleine begriffliche Unterscheidung, die aber nach allem bisher Gesagten schnell abgehandelt werden kann.

Lokal vs. Attribut vs. Parameter



```
public class X {  
    private int i;  
    public void m ( double d ) {  
        char c1;  
        final char C2 = 'a';  
        .....  
    }  
}
```

i: Attribut von Klasse X

d: Parameter von Methode m

c1: lokale Variable in Methode m

C2: lokale Konstante in Methode m

Dazu reicht sogar schon dieses eine kleine, rein illustrative Beispiel. Was Attribute von Klassen und Parameter von Methoden sind, wissen Sie ja. Lokale Variable wie hier c1 beziehungsweise lokale Konstanten wie c2 mit final sind diejenigen Variablen und Konstanten, die im Rumpf einer Methode definiert werden – eben lokal in dieser Methode. Das war es auch schon.

Scope von Definitionen von Identifiern

Einen wichtigen Punkt haben wir bisher nur am Rande gestreift, jetzt betrachten wir ihn systematisch: Wo im Quelltext ist welcher Identifier gültig? Für Identifier, die Variable oder Attribute bezeichnen, bedeutet das: Wo im Quelltext kann man lesend und schreibend darauf zugreifen; bei Konstanten natürlich entsprechend *nur* lesend. Bei Identifiern, die Methoden bezeichnen, bedeutet das: Wo überall kann diese Methode aufgerufen werden? Bei Namen von Klassen, Interfaces und Enumerationen bedeutet es: Wo überall können Variable und Konstanten dieses Typs eingerichtet werden? In jedem Fall bezeichnet der Fachbegriff *Scope* diesen Bereich im Quelltext.

Sowohl in Java als auch in anderen Sprachen gilt die Regel: Der Scope der Definition eines Identifiers lässt sich eindeutig aus dem Quelltext selbst erschließen. Analog zum statischen versus dynamischen Typ einer Variablen, sagen wir daher, der Scope einer Definition ist *statisch*.

Name einer Klasse oder eines Interface:

- **Überall wo Package importiert**
- **Alternativ voll qualifizierter Name**
 - **Packagehierarchie vorangestellt**

Diese Folie ist im Wesentlichen eine Wiederholung von weiter vorne in diesem Kapitel. Eine Klasse oder ein Interface oder eine Enumeration muss importiert werden, falls es ohne die Angabe des Packages, Subpackages und so weiter verwendet werden soll.

Scope von Definitionen



Attribut / Methode von Klasse:

- **bei private: überall innerhalb der Klasse**
- **ohne explizites Zugriffsrecht: *zusätzlich* auch im gesamten Package**
- **bei protected: *zusätzlich* auch in abgeleiteten Klassen**
- **bei public: überall**

Wo überall ist nun ein Attribut beziehungsweise eine Methode einer Klasse oder eines Interface sichtbar? Bei Klassen muss zwischen den verschiedenen Zugriffsrechten unterschieden werden. Das hatten wir schon im Kapitel zu objektorientierter Abstraktion betrachtet, Abschnitt zur Vererbung von Attributen. Die Auflistung auf dieser Folie ist also ebenfalls im Wesentlichen eine Wiederholung.

Bei Interfaces trifft bekanntlich nur der letzte Fall zu: public.

Scope von Definitionen



Attribut / Methode von Klasse:

- bei **private**: **überall** innerhalb der Klasse
- ohne explizites Zugriffsrecht: *zusätzlich* auch im gesamten Package
- bei **protected**: *zusätzlich* auch in abgeleiteten Klassen
- bei **public**: überall

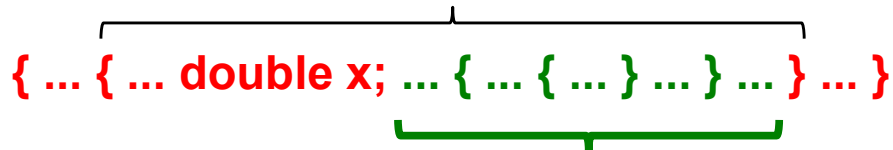
Ein Wort noch zur Verwendbarkeit innerhalb der eigenen Klasse:
„überall“ meint tatsächlich „überall“, also auch vor der Einführung des Identifiers.

Scope von Definitionen

```
public class X {  
    public double m ( int n ) {  
        double x = n / 3.0;  
        return x;  
    }  
}
```

Innerhalb einer Methode sind zwei weitere Entitäten wichtig, die durch Identifier bezeichnet werden: Parameter wie hier *n* und lokale Variable wie hier *x*. Der Scope eines Parameters ist der gesamte Methodenrumpf.

Scope von Definitionen

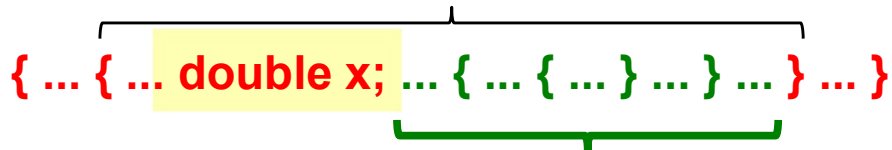


```
{ ... { ... double x; ... { ... { ... } ... } ... } ... }
```

Der Scope einer lokalen Variablen lässt sich gut anhand eines schematischen Beispiels wie hier zeigen. Die obere, schwarze horizontale Klammer umreißt das innerste Klammerpaar, in dem der Identifier `x` definiert ist. Der grüne Bereich, der auch durch die untere, grüne horizontale Klammer umrissen ist, ist der Scope der Definition des Identifiers `x`. Allgemein ist der Scope einer lokalen Variable genau der Bereich ab der Definition bis zum Ende des Bereichs, der durch das innerste umfassende Klammerpaar eingegrenzt ist. Dies schließt auch alle Teilbereiche ein, die durch weitere geschweifte Klammerpaare begrenzt werden, beispielsweise Rümpfe von Schleifen.

Zugriff auf `x` außerhalb des grünen Bereichs führt zu einer Fehlermeldung des Compilers nebst Abbruch der Übersetzung.

Scope von Definitionen



```
{ ... { ... double x; ... { ... { ... } ... } ... } ... }
```

Erinnerung: Allerdings gibt es noch eine offenkundig sinnvolle Sonderregel: Auf eine lokale Variable darf nicht *lesend* zugegriffen werden, bevor das erste Mal *schreibend* darauf zugegriffen wurde. Wird die Variable wie hier nicht initialisiert, dann heißt das: Auf sie darf nicht lesend zugegriffen werden, bevor es eine Zuweisung an sie gab.

Denn wie wir gesehen haben, werden lokale Variable im Gegensatz zu Attributen und Arraykomponenten nicht automatisch mit dem *Nullwert* oder irgendeinem anderen Wert initialisiert, sondern enthalten erst einmal einen Zufallswert.

Bei Parametern kann das natürlich nicht passieren; die werden durch den Methodenaufruf mit den aktuellen Parameterwerten initialisiert.

Scope von Definitionen



```
for ( int i = 0; i < n; i++ )  
    sum += i;  
  
for ( int i = m; i > 0; i-- ) {  
    prod *= i;  
    System.out.println ( prod );  
}
```

Eine Besonderheit bei der for-Schleife haben wir schon kennen gelernt: Vor dem ersten Semikolon im Kopf der for-Schleife können auch lokale Variable definiert werden. Der Scope einer solchen lokalen Variable ist die for-Schleife selbst, also ihr Kopf und ihr Rumpf. Und zwar egal, ob der Rumpf eine einfache Anweisung wie oben oder eine Blockanweisung wie unten ist.

In diesen beiden for-Schleifen werden zwei verschiedene Variable mit demselben Namen i definiert.

Zugriff auf den Identifier i außerhalb der beiden for-Schleifen führt wieder zu einer Fehlermeldung des Compilers nebst Abbruch der Übersetzung.


```
for ( double x : a )  
    sum += x;
```

Natürlich gilt dasselbe für die verkürzte Form der for-Schleife: `x` kann nur innerhalb der for-Schleife verwendet werden.

***Vorgriff:* Im Kapitel 05 zur Fehlerbehandlung werden wir noch eine weitere Scope-Regel kennen lernen, die sich auf die Art von Anweisungsblöcken beziehen, die wir dort *catch-Blöcke* nennen werden. Die Struktur ist so ähnlich wie bei for-Schleifen.**

Scope von Definitionen

```
public class X {  
    private double x;  
    public X ( double x ) {  
        this.x = x;  
    }  
    public void setX ( double x ) {  
        this.x = x;  
    }  
}
```

Eine Sache, die wir schon gesehen haben, ist zu beachten: Parameter und lokale Variable einer Methode können genauso heißen wie Attribute der Klasse, zu der die Methode gehört. Wir hatten dies schon im Abschnitt zu this und super. Im Beispiel auf dieser Folie haben beide Methoden einen Parameter namens x, und die Klasse hat ein Attribut namens x.

Scope von Definitionen

```
public class X {  
    private double x;  
    public X ( double x ) {  
        this.x = x;  
    }  
    public void setX ( double x ) {  
        this.x = x;  
    }  
}
```

Wird nur x alleine verwendet, dann ist das der Parameter beziehungsweise die lokale Variable.

Scope von Definitionen

```
public class X {  
    private double x;  
    public X ( double x ) {  
        this.x = x;  
    }  
    public void setX ( double x ) {  
        this.x = x;  
    }  
}
```

Um das *Attribut* anzusprechen, muss Schlüsselwort **this** davor geschrieben werden, durch einen Punkt verknüpft.

Scope von Definitionen



```
public void m ( int n ) {      int i;
    double n;                  for ( int i = 0; .....
}

int n;                          for ( int i = 0; i < n; i++ ) {
while ( true ) {                long i;
    long n;                      .....
    .....                       }
}
```

Abgesehen von diesem einen Fall dürfen sich die Scopes verschiedener Definitionen desselben Identifiers nicht überschneiden. Auf dieser Folie sehen Sie zum Ende dieses Abschnitts vier Beispiele, die allesamt nicht durch den Compiler gehen.

Scope von Definitionen



```
public void m ( int n ) {      int i;
    double n;                 for ( int i = 0; .....
}

int n;                        for ( int i = 0; i < n; i++ ) {
while ( true ) {              long i;
    long n;                   .....
    .....                    }
}
```

Parameter und lokale Variable derselben Methode dürfen nicht mit demselben Identifier bezeichnet sein. Dabei ist es wie auch in den folgenden Beispielen egal, ob der Datentyp derselbe oder so wie hier unterschiedlich ist.

Scope von Definitionen



```
public void m ( int n ) {           int i;
    double n;                       for ( int i = 0; .....
}

int n;                               for ( int i = 0; i < n; i++ ) {
while ( true ) {                     long i;
    long n;                          .....
    .....                           }
}
```

Im Scope einer lokalen Variable darf derselbe Identifier nicht nochmals für eine andere lokale Variable verwendet werden.

Scope von Definitionen



```
public void m ( int n ) {  
    double n;  
}  
  
int n;  
while ( true ) {  
    long n;  
    .....  
}  
  
int i;  
for ( int i = 0; .....  
  
for ( int i = 0; i < n; i++ ) {  
    long i;  
    .....  
}
```

Ein schon gültiger Identifier darf nicht in einer for-Schleife noch einmal definiert werden.

Scope von Definitionen



```
public void m ( int n ) {      int i;
    double n;                  for ( int i = 0; .....
}

int n;                          for ( int i = 0; i < n; i++ ) {
while ( true ) {                long i;
    long n;                      .....
    .....                       }
}
```

Und natürlich auch nicht umgekehrt.

Damit beschließen wir den Abschnitt zu Scopes und das gesamte Kapitel.