

# 3. Gültigkeit von Definitionen

Themen dieses Kapitels:

- Definition und Bindung von Bezeichnern
- Verdeckungsregeln für die Gültigkeit von Definitionen
- Gültigkeitsregeln in Programmiersprachen

# Definition und Bindung

Eine **Definition** ist ein Programmkonstrukt, das die **Beschreibung eines Programmgegenstandes an einen Bezeichner bindet**.

**Programmkonstrukt:** zusammengehöriger Teil (Teilbaum) eines Programms

z. B. eine Deklaration `int i;`, eine Anweisung `i = 42;` Ausdruck `i+1`

**Programmgegenstand:** wird im Programm beschrieben und benutzt

z. B. die Methode `main`, der Typ `string`, eine Variable `i`, ein Parameter `args`

Meist legt die Definition Eigenschaften des **Programmgegenstandes** fest,  
z. B. den Typ:

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

# Statische und dynamische Bindung

Ein Bezeichner, der in einer **Definition** gebunden wird, tritt dort **definierend** auf; an anderen Stellen tritt er **angewandt** auf.

Definierendes und angewandtes Auftreten von Bezeichnern kann man meist **syntaktisch unterscheiden**, z. B.

```
static int ggt (int a, int b)
{ ...
  return ggt(a % b, b);
...
}
```

Regeln der Sprache entscheiden, in welcher **Definition** ein **angewandtes** Auftreten eines Bezeichners gebunden ist.

## Statische Bindung:

Gültigkeitsregeln entscheiden die Bindung am **Programmtext**, z. B.

```
{ float a = 1.0;
  { int a = 2;
    printf ("%d", a);
  }
}
```



statische Bindung im Rest dieses Kapitels und in den meisten Sprachen, außer ...

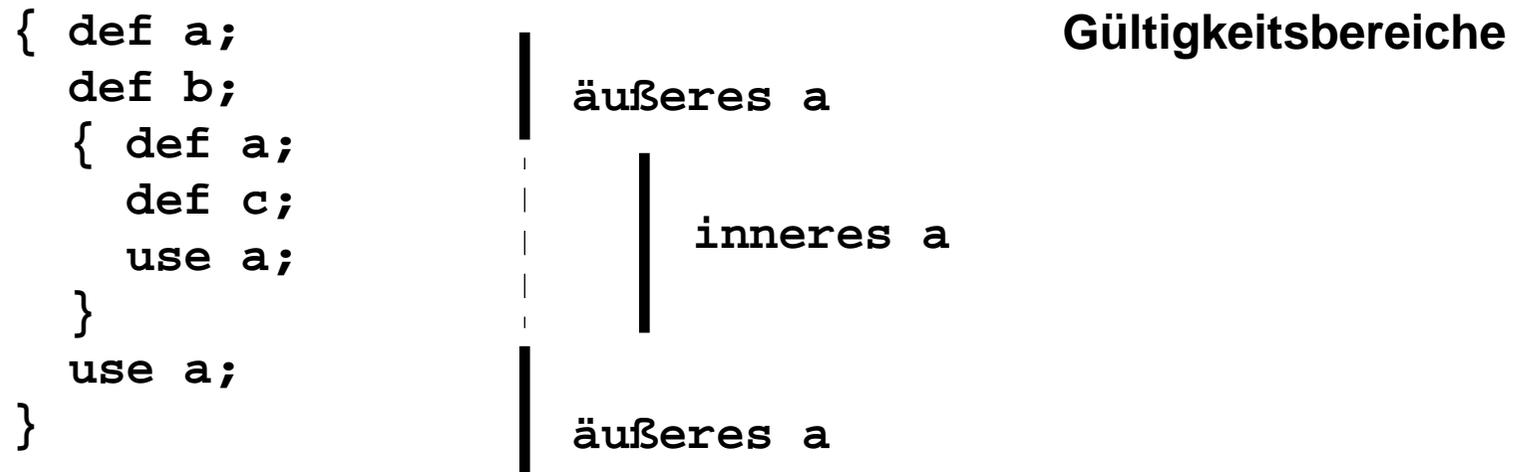
## Dynamische Bindung:

Wird bei der **Ausführung des Programms** entschieden:  
Für einen angewandten Bezeichner **a** gilt die zuletzt für **a** **ausgeführte** Definition.

dynamische Bindung  
in Lisp und einigen Skriptsprachen

# Gültigkeitsbereich

Der **Gültigkeitsbereich (scope)** einer Definition  $D$  für einen Bezeichner  $b$  ist der Programmabschnitt, in dem angewandte Auftreten von  $b$  an den in  $D$  definierten Programmgegenstand gebunden sind.



In **qualifizierten Namen**, können Bezeichner auch außerhalb des Gültigkeitsbereiches ihrer Definition angewandt werden:

```
Thread.sleep(1000); max = super.MAX_THINGS;
```

`sleep` ist in der Klasse `Thread` definiert, `MAX_THINGS` in einer Oberklasse.

# Verdeckung von Definitionen

In Sprachen mit geschachtelten Programmstrukturen kann eine Definition eine andere für den gleichen Bezeichner **verdecken** (**hiding**).

Es gibt **2 unterschiedliche Grundregeln** dafür:

**Algol-Verdeckungsregel** (in Algol-60, Algol-68, Pascal, Modula-2, Ada, Java s. u.):

Eine Definition gilt im kleinsten sie umfassenden Abschnitt **überall**, ausgenommen darin enthaltene Abschnitte mit einer Definition für denselben Bezeichner.

oder operational formuliert:

Suche vom angewandten Auftreten eines Bezeichners **b** ausgehend nach außen den kleinsten umfassenden Abschnitt mit einer Definition für **b**.

**C-Verdeckungsregel** (in C, C++, Java):

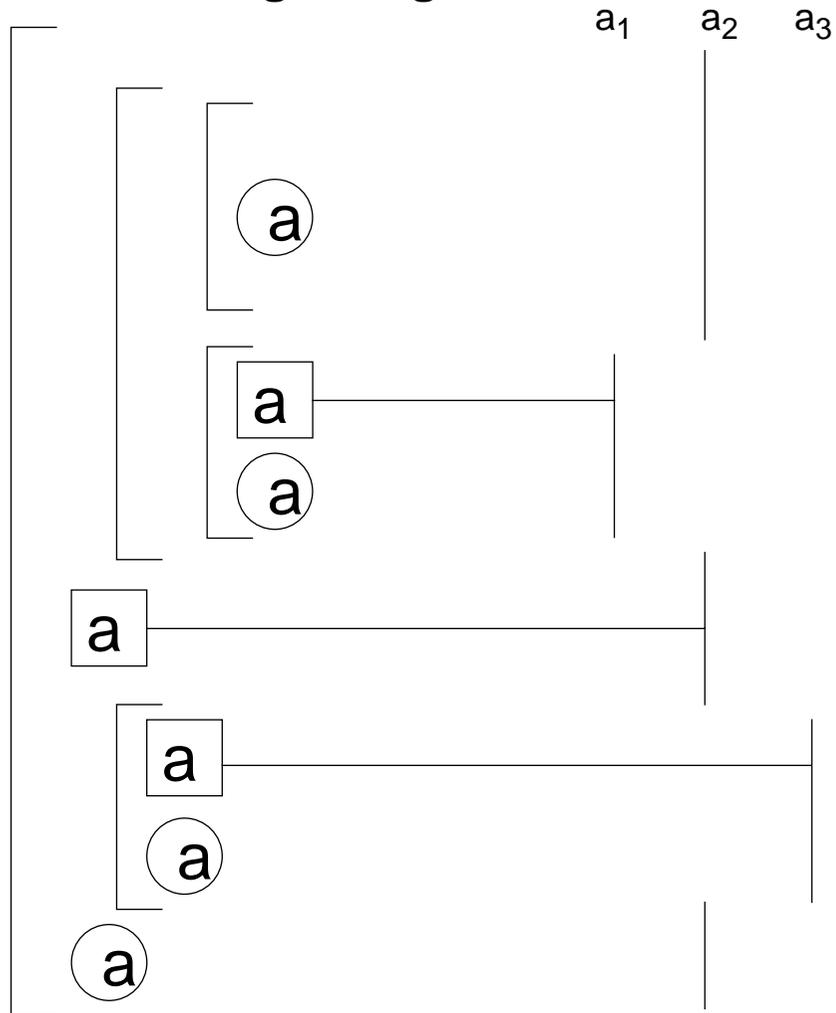
Die Definition eines Bezeichners **b** gilt **von der Definitionsstelle** bis zum Ende des kleinsten sie umfassenden Abschnitts, **ausgenommen die Gültigkeitsbereiche von Definitionen für b** in darin enthaltenen Abschnitten.

Die **C-Regel** erzwingt **definierendes** vor **angewandtem** Auftreten.

Die **Algol-Regel** ist einfacher, toleranter und vermeidet Sonderregeln für notwendige Vorwärtsreferenzen.

# Beispiele für Gültigkeitsbereiche

## Algol-Regel



## Symbole:

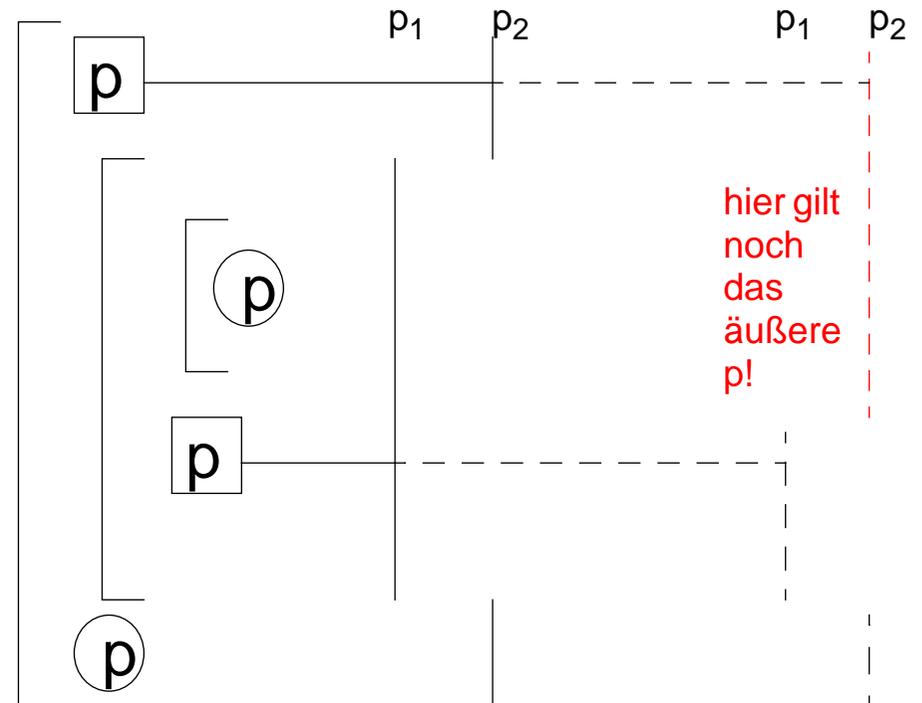
Abschnitt

$a$  Definition

$a$  Anwendung

## Algol-Regel

## C-Regel



# Getrennte Namensräume

In manchen Sprachen werden die Bezeichner für Programmgegenstände bestimmter Art jeweils einem **Namensraum** zugeordnet

z. B. in **Java** jeweils ein Namensraum für

- Packages, Typen (Klassen und Interfaces), Variable (lokale Variable, Parameter, Objekt- und Klassenvariable), Methoden, Anweisungsmarken

Gültigkeits- und Verdeckungsregeln werden **nur innerhab eines Namensraumes** angewandt - nicht zwischen verschiedenen Namensräumen.

Zu welchem Namensraum ein Bezeichner gehört, kann am **syntaktischen Kontext** erkannt werden. (In Java mit einigen zusätzlichen Regeln)

Eine  
Klassendeklaration  
nur für Zwecke der  
Demonstration:

```
class Multi {  
    Multi () { Multi = 5;}  
    private int Multi;  
    Multi Multi (Multi Multi) {  
        if (Multi == null)  
            return new Multi();  
        else return Multi (new Multi ());  
    }  
}
```

Typ

Variable

Methode

# Gültigkeitsbereiche in Java

## Package-Namen:

sichtbare Übersetzungseinheiten

## Typnamen:

in der ganzen Übersetzungseinheit, Algol-60-Verdeckungsregel

## Methodennamen:

umgebende Klasse, Algol-60-Verdeckungsregel, aber  
Objektmethoden der Oberklassen werden überschrieben oder überladen - nicht verdeckt

## Namen von **Objekt- und Klassenvariablen:**

umgebende Klasse, Algol-60-Verdeckungsregel,  
Objekt- und Klassenvariable können Variable der Oberklassen verdecken

## Parameter:

Methodenrumpf, (dürfen nur durch innere Klassen verdeckt werden)

## Lokale Variable:

Rest des Blockes (bzw. bei Laufvariable in for-Schleife: Rest der for-Schleife),  
C-Verdeckungsregel (dürfen nur durch innere Klassen verdeckt werden)

## Terminologie in Java:

*shadowing* für *verdecken* bei Schachtelung, *hiding* für *verdecken* beim Erben

# Beispiele für Gültigkeitsbereiche in Java

```

                                A B m mm cnt p f
class A
{
    void m (int p)
    { cnt += 1;
      float f;
      ...
    }
    B mm ()
    { return new B();
    }
    int cnt = 42;
}

class B
{
    ...
}

```

```

class Ober
{ int k;
  ...
}

class Unter extends Ober
{ int k;
  void m ()
  { k = 5;
  }
  void g (int p)
  { int k = 7;
    k = 42;
    for (int i = 0;
          i < 10; i++)
    {
      int k; // verboten
      ...
    }
  }
}

```

# Innere Klassen in Java: Verdeckung von lokalen Variablen

```

class A
{ char x;

  void m ()
  { int x;

    class B
    {
      void h ()
      { float x;
        ...
      }
      ...
    }
    ...
  }
}

```

char    int    float  
 x        x        x

The diagram illustrates the scope of three variables: `char x`, `int x`, and `float x`. Vertical lines represent the active scope of each variable. A black line for `char x` spans the entire class `A`. A red line for `int x` spans the `m()` method of `A` and the `h()` method of `B`. A shorter red line for `float x` spans only the `h()` method of `B`. Ellipses (...) indicate that the scope continues through other parts of the code not shown.

Innere Klasse B:  
 Lokale Variable `float x` in `h`  
 verdeckt  
 lokale Variable `int x` in `m`  
 der äußeren Klasse

# Gültigkeitsregeln in anderen Programmiersprachen

## C, C++:

grundsätzlich gilt die **C-Regel**;  
für Sprungmarken gilt die **Algol-Regel**.

## Pascal, Ada, Modula-2:

grundsätzlich gilt die **Algol-Regel**.  
Aber eine **Zusatzregel** fordert:

```
void f () {
    ...
    goto finish;
    ...
finish: printf (...);
}
```

Ein **angewandtes Auftreten** eines Bezeichners darf **nicht vor seiner Definition** stehen.

Davon gibt es dann in den Sprachen unterschiedliche **Ausnahmen**, um wechselseitig rekursive Definitionen von Funktionen und Typen zu ermöglichen.

### Pascal:

```
type ListPtr = ^ List;
   List = record
       i: integer;
       n: ListPtr
   end;
```

### C:

```
typedef struct _el *ListPtr;
typedef struct _el
{ int i; ListPtr n;} Elem;
```

### Pascal:

```
procedure f (a:real) forward;

procedure g (b:real)
begin ... f(3.5); ... end;

procedure f (a:real)
begin ... g(7.5); ... end;
```

# Zusammenfassung zum Kapitel 3

Mit den Vorlesungen und Übungen zu Kapitel 3 sollen Sie nun Folgendes können:

- Bindung von Bezeichnern verstehen
- Verdeckungsregeln für die Gültigkeit von Definitionen anwenden
- Grundbegriffe in den Gültigkeitsregeln von Programmiersprachen erkennen