# 9
# Pakete, abstrakte Klassen, Interfaces

- Pakete
- Nutzen von verteilten Ressourcen
- Zugriffskontrolle
- Der Modifizierer **final**
- Innere und anonyme Klassen
- Abstrakte Klassen
- Interfaces

# 9.1 Pakete (Packages)

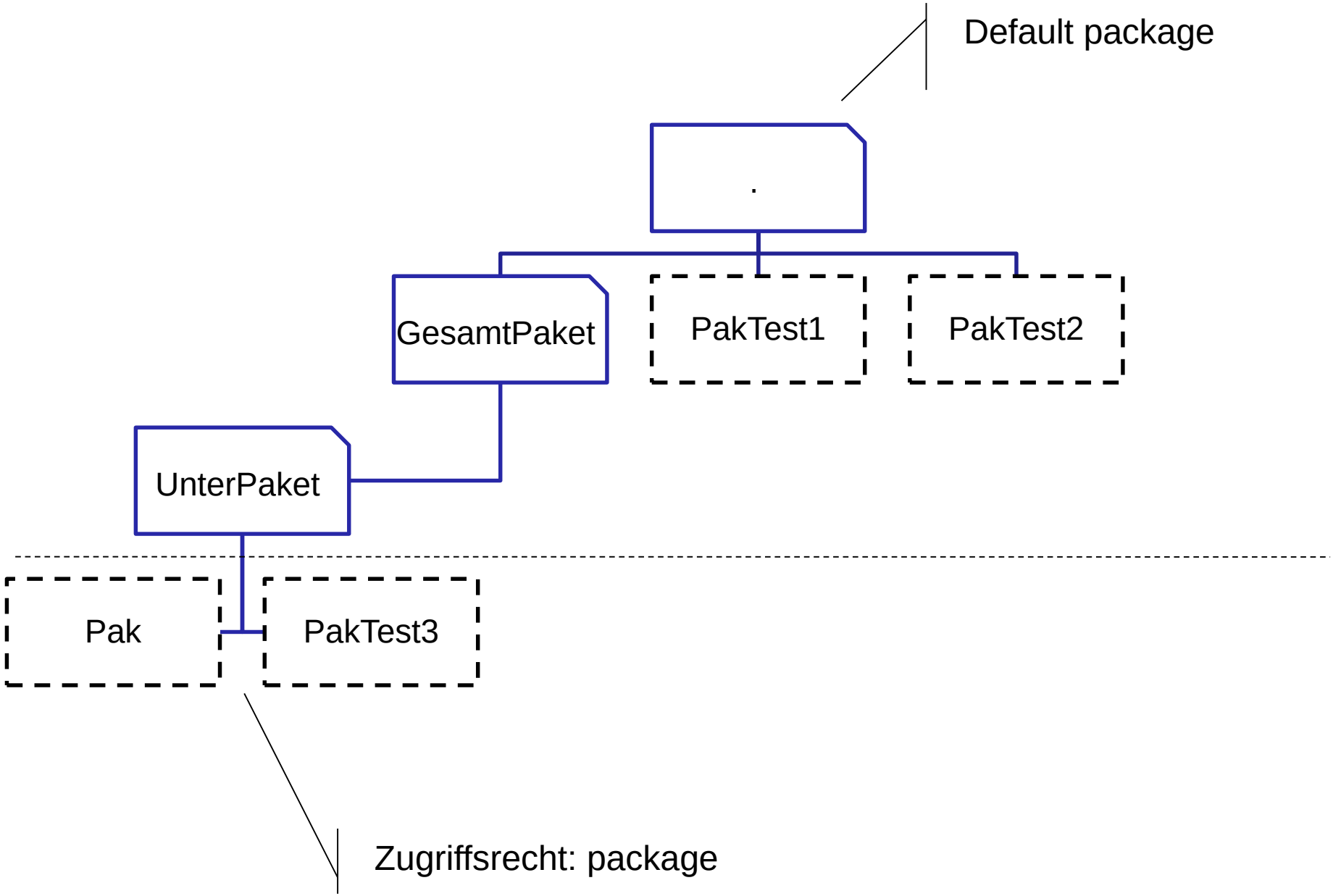
- Gruppierungen von Klassen mittels Deklaration der Paket-Zugehörigkeit durch entsprechende Angabe zu Beginn einer Übersetzungseinheit
- Beispiele

```
package meinErstesPaket;
public class meineErsteKlasse {
   . . .
}
```

```
package meinGesamtPaket.meinZweitesPaket;
public class meineZweiteKlasse {
   . . .
}
```

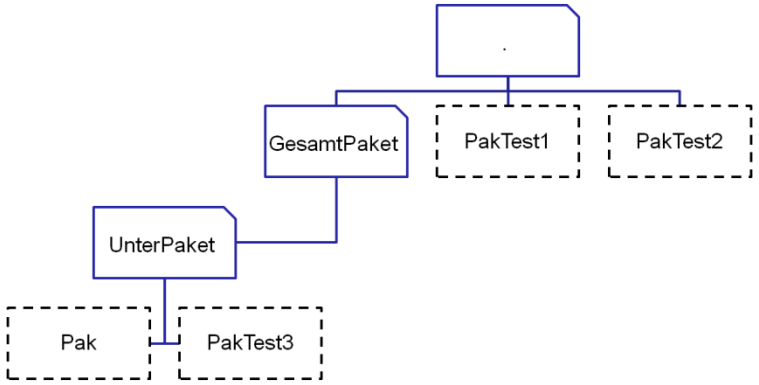
- Verwendung von Elementen anderer Klassen in fremden Paketen
  - *mit* Paket- und Klassennamen (z. B. **`a.b.c.x.y.Kla.klaMeth();`**)
  - *ohne*, nach vorheriger **`import`**-Anweisung (z. B. **`import a.b.c.x.y.Kla;`**)
- Bei einem Import können alle Klassen (**`*`**) oder einzelne Klassen bzw. Interfaces eines Pakets bereitgestellt werden. (Vorsicht: **`*`** steht nicht für Unterpakete!)

# Pakete


Default package
.
GesamtPaket
PakTest1
PakTest2
UnterPaket
Pak
PakTest3
Zugriffsrecht: package

# Pakete

```
package GesamtPaket.UnterPaket;
class Pak {
  void test() {
    System.out.println("Pak.test");
  }
}
```




```
class PakTest1 {
  public static void main(String[] args) {
    new Pak().test();  // Fehler: Klasse unbekannt
  }
}
```

```
class PakTest2 {
  public static void main(String[] args) {
    new GesamtPaket.UnterPaket.Pak().test();  // Fehler
  }                                           // Klasse Pak ist nicht public
}
```

```
package GesamtPaket.UnterPaket;
class PakTest3 {
  public static void main(String[] args) {
    new Pak().test();  // ok!
  }
}
```

# Ergänzung: Verwendung externer Klassen

- Problem:  Nutzung von Klassen, die an verschiedenen Stellen im System liegen
- Grundsätzliche Frage: Wie werden Ressourcen im System gefunden
- Lösung: Umgebungsvariablen, die zentrale Informationen für verschiedene Anwendungen speichern (Suchpfade, Default-Drucker, …)

# Aufruf eines Programms aus der Shell

- Aufruf eines Programms, z.B. *emacs* oder *cp* aus dem aktuellen Verzeichnis (z.B. `~/home/urost/`):

```
>   emacs Hello.java &
>   cp test.txt test2.txt
```

- Frage: wo liegt das ausführbare Programm und woher weiß die Laufzeitumgebung, wo sie danach suchen soll?
  - Laufzeitumgebung sucht nach Programmen nur in den Verzeichnissen, die in der Umgebungsvariablen PATH gespeichert sind.
  - Ausgabe des aktuellen Pfads mit **`echo $PATH`** (unter Linux)
  - Liegt ein Programm nicht in einem der dort aufgeführten Verzeichnisse, wird es nicht gefunden

# Umgebungsvariablen

- Um Programme, die in einer Shell aufgerufen werden, im Dateibaum zu lokalisieren, gibt es sogenannte Umgebungsvariablen, z.B. **`PATH`** die eine Liste der zu durchsuchenden Pfade enthalten
  - Die Pfade können interaktiv mit dem **`export`**-Befehl in der Shell gesetzt werden (wenn Sie z.B. nur temporär benötigt werden). Sie gelten dann nur in diesem Terminal und denen, die daraus erzeugt/aufgerufen werden.
  - Ansehen einer solchen Variable mit dem echo-Befehl, z.B. **`echo $PATH`**
  - Die Befehle zum setzen von Umgebungsvariablen können auch in einer speziellen Datei gespeichert sein, die jedesmal beim Aufruf einer Shell (beim Öffnen eines Terminals) zu Beginn geladen und ausgeführt wird.
  - Für die **`bash`** ist dies die Dateil **`.profile`** (z.T. auch **`.bashrc),`** die normalerweise direkt im **`HOME`**-Verzeichnis liegt

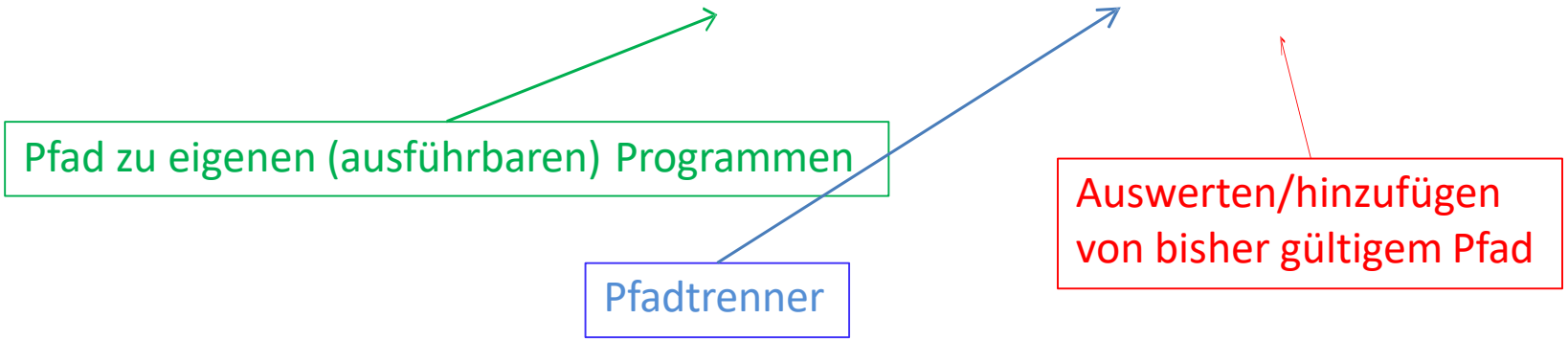
# Beispiel für `.profile` Datei

```
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
# exists.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.
#
export JAVA_HOME=/usr/lib/jdk1.8.0_121
export CATALINA_HOME=$HOME/sw/apache-tomcat-8.0.30

# IOTools
export CLASSPATH="$HOME/java-lib:$CLASSPATH"

# JDOM
export CLASSPATH="$HOME/java/jdom/jdom-2.0.6.jar:$CLASSPATH"
```

# Umgebungsvariablen (cont.)

- Wichtige Umgebungsvariablen:
  - **`PATH`** (für ausführbare Binärdateien)
  - **`LD_LIBRARY_PATH`** (für dynamisch dazuzubindende Bibliotheken
  - **`CLASSPATH`** (für Java Klassen)
- Befehl zum setzen einer PATH-Variable (Beispiel)

```
export PATH=$HOME/project/java:$PATH
```




- Aktueller Pfad kann auch in PATH aufgenommen werden
  - `export PATH=./:$PATH`

# CLASSPATH

- Sind Klassen im Default-Package definiert (keine Package-Angabe), werden Sie nur dann gefunden, wenn der CLASSPATH das Verzeichnis, in dem Sie liegen, enthält:
- Beispiel:
  - Klasse **`IOTools`** (**`IOTools.class`**) liegt im Verzeichnis **`/home/urost/java-lib`**
  - Klasse **`TestIO`** liegt im Verzeichnis **`/home/urost/project/test`**
  - Wenn die Klasse **`IOTools`** in der Klasse **`TestIO`** benötigt wird, dann muss der Aufruf von Compiler und Virtueller Maschine mit der Option für den CLASSPATH aufgerufen werden

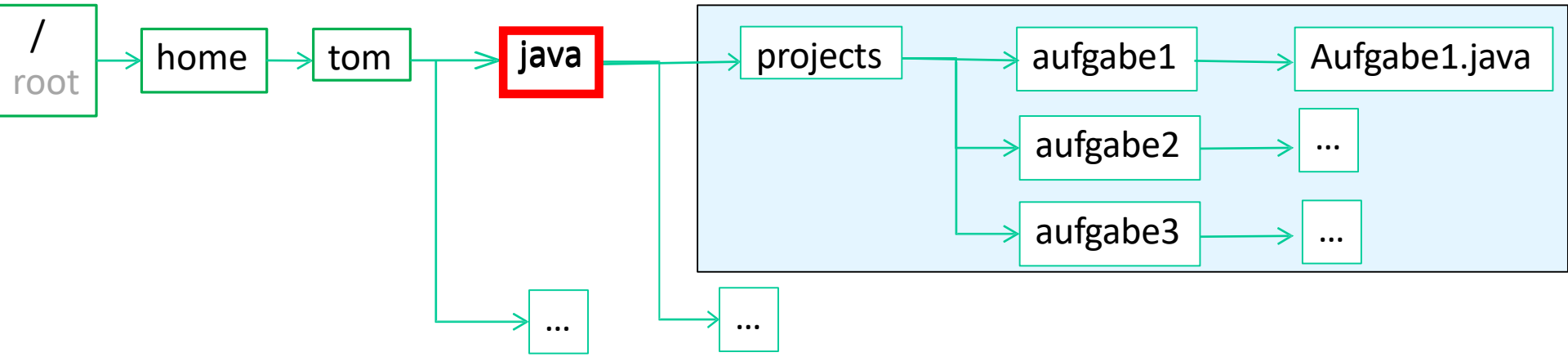
    ```
    javac -cp /home/urost/java-lib TestIO.java

    java -cp /home/urost/java-lib:$CLASSPATH TestIO
    ```

  - oder vorher der CLASSPATH um das Verzeichnis **`/home/urost/java-lib`** ergänzt werden.

    ```
    export CLASSPATH=/home/urost/java-lib:$CLASSPATH
    ```

voreingestellter Klassenpfad

# Compilieren und Ausführen von Klassen in Packages




- Aufgabe_1.java enthält Anweisung **`package projects.aufgabe1;`**
- CLASSPATH ergänzen um **`/home/tom/java`**
- Übersetzen (z.B. hier in **`/home/tom`**):

  **`javac java/projects/aufgabe1/Aufgabe_1.java`**
- Ausführen (überall):

  **`java projects.aufgabe1.Aufgabe_1`**
- Merke: Paket-Hierarchie spiegelt Verzeichnis-Hierarchie wieder (auch in jar-Files)

# 9.2 Zugriffskontrolle

- Auf alle Variablen und Methoden einer Klasse kann generell überall innerhalb der definierenden Klasse zugegriffen werden.
- Von anderen Klassen aus kann auf diese Variablen und Methoden nur zugegriffen werden, wenn in der definierenden Klasse der Zugriff explizit gewährt wird.
- Den Typ des Zugriffs legen die Modifikatoren **`private`**, **`protected`** oder **`public`** fest. Ohne Angabe eines Modifikators ist der Zugriff vom Typ package.
- **`public`** ***erlaubt*** den Zugriff von allen anderen Klassen aus.
- **`private`** ***untersagt*** den Zugriff von allen anderen Klassen aus.
- package (also ohne Modifikator) ***erlaubt*** den Zugriff von anderen Klassen des gleichen Pakets aus.
- **`protected`** erlaubt den Zugriff von anderen Klassen desselben Pakets aus und von abgeleiteten Klassen (auch in anderen Paketen) aus. Allerdings ist in abgeleiteten Klassen in anderen Paketen der Zugriff ***nur für geerbte*** Variablen und Methoden erlaubt.

# Zugriffskontrolle

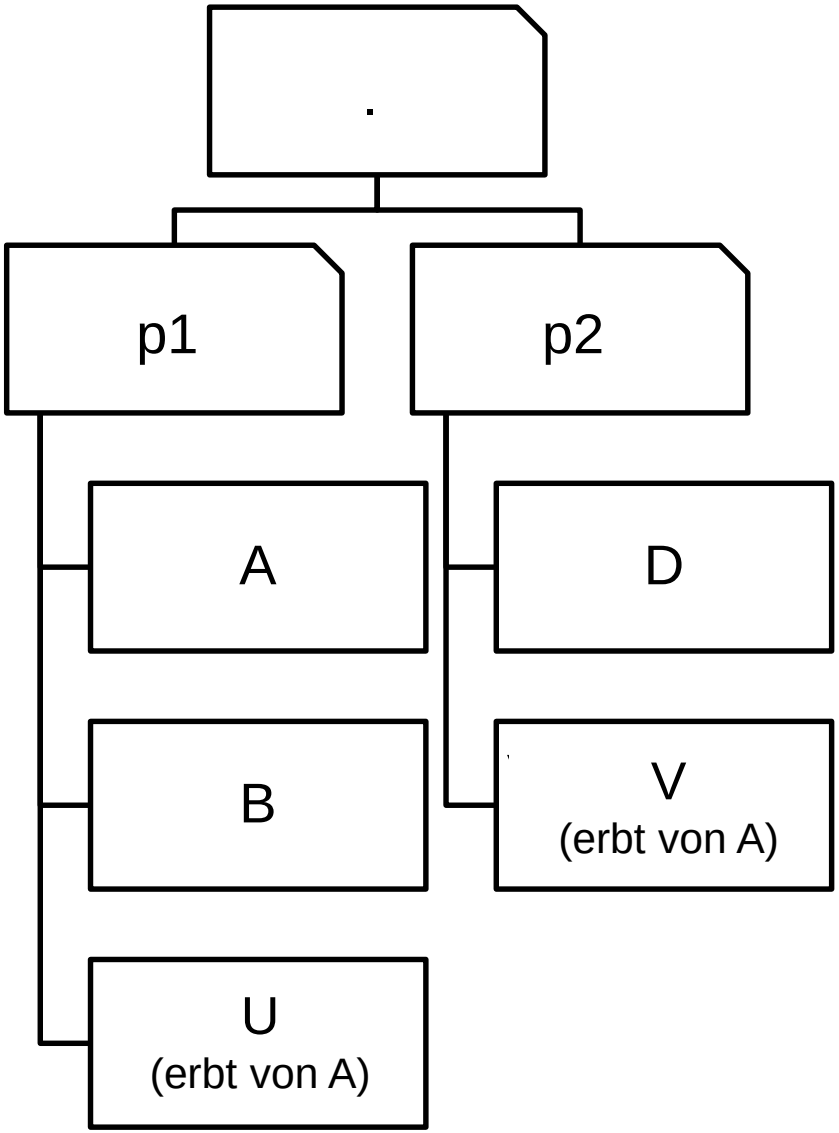
- **Übersicht der erlaubten Zugriffe (durch x markiert) auf Variablen und Methoden (mit dem angegebenen Modifizierer) der Klasse `A`**

  Die Klassen **`A`**, **`B`** und **`U`** liegen im Paket **`p1`**, **`U`** erbt von **`A`**
  Die Klassen **`D`** und **`V`** liegen im Paket **`p2`**, **`V`** erbt von **`A`**

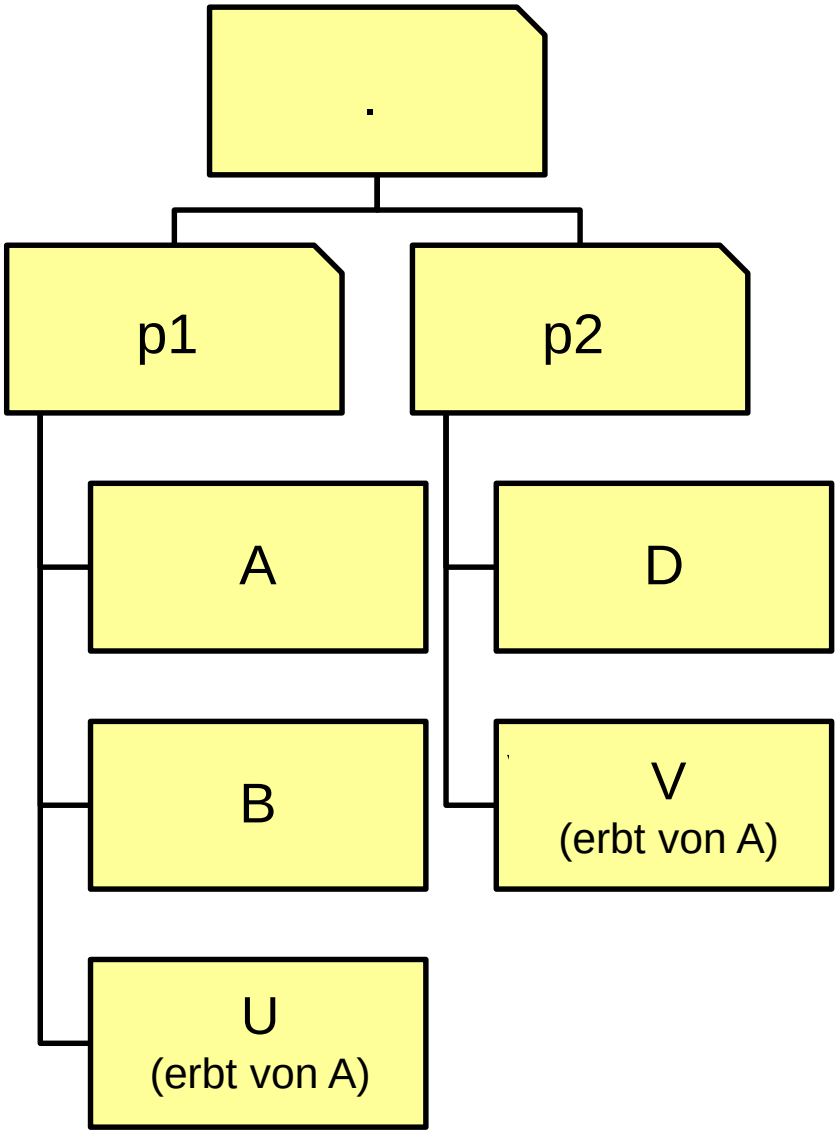
| in Klasse / Modifizierer | `A` | `B` | `U` | `V` | `D` |
|---|---|---|---|---|---|
| `private` | X | | | | |
| `(package)` | X | X | X | | |
| `protected` | X | X | X | X(*) | |
| `public` | X | X | X | X | X |

(*) Der Zugriff ist nur für eigene, von **`A`** geerbte Variablen und Methoden erlaubt!

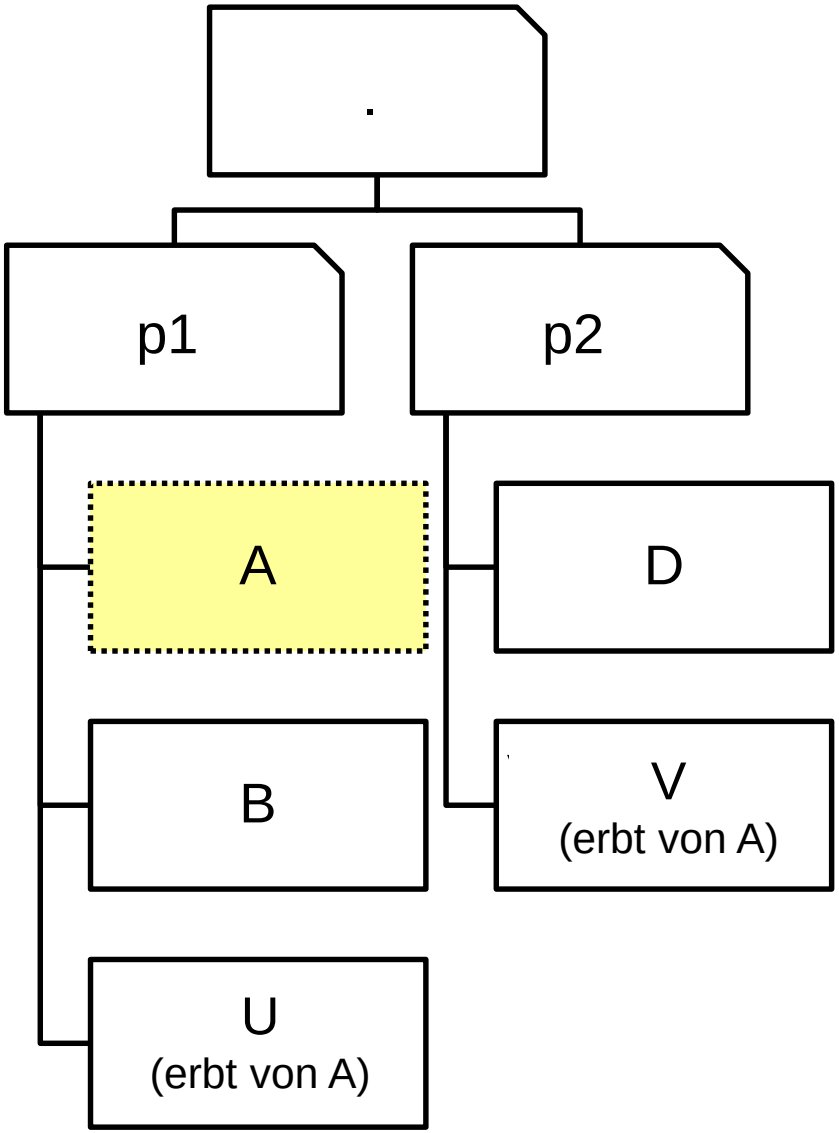
# Zugriffskontrolle: Beispiel für Zugriffe auf Elemente von A



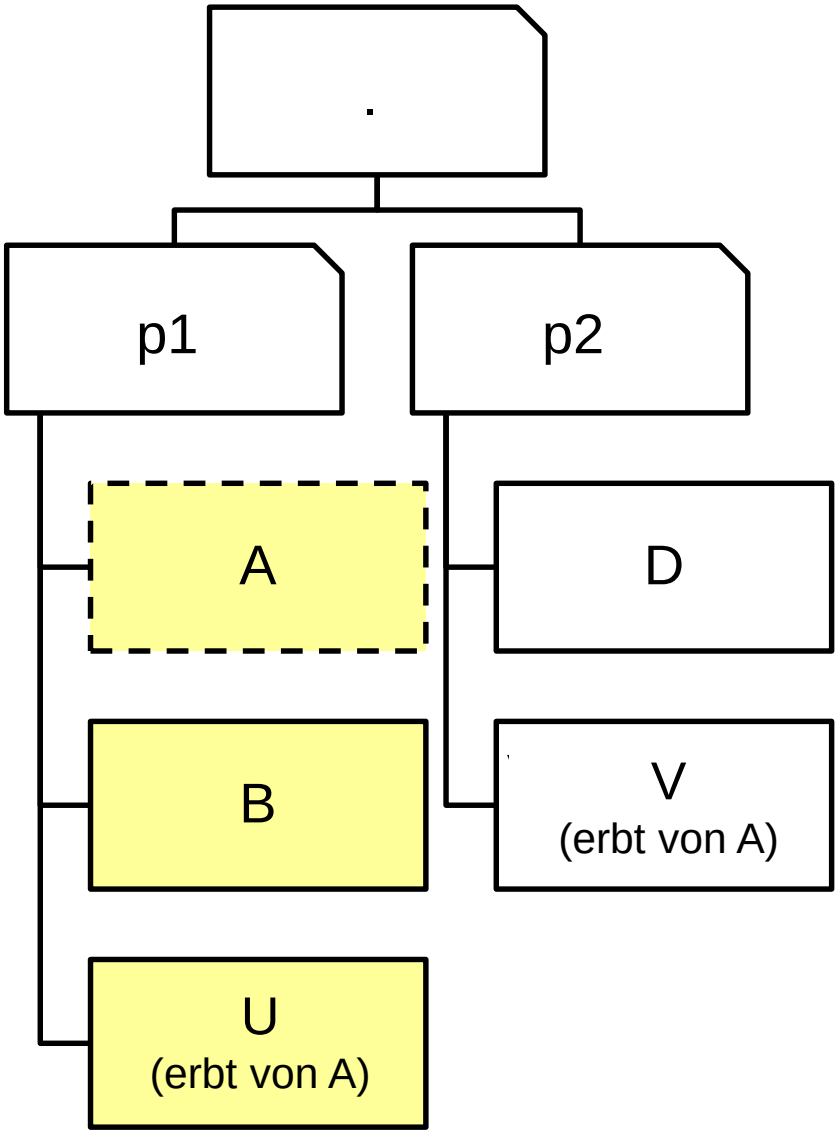
# Zugriff auf Element der Klasse A mit Modifizierer `public`



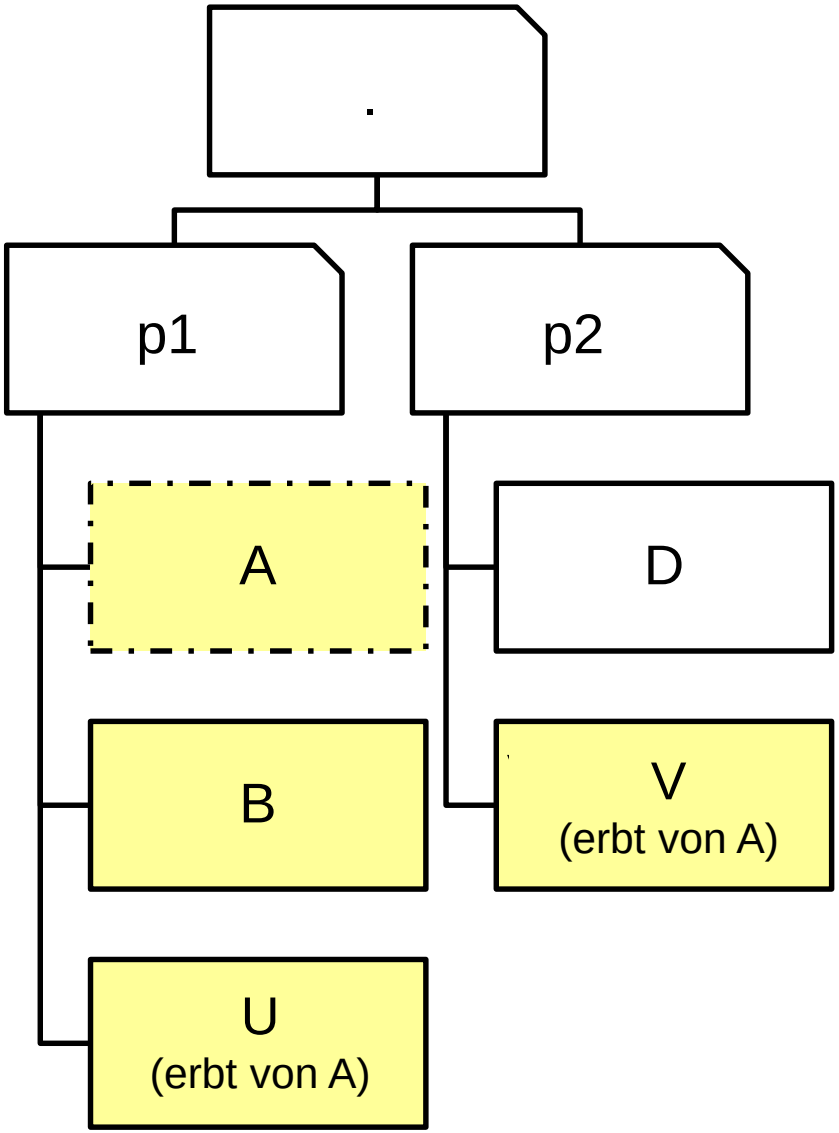
# Zugriff auf Element der Klasse A mit Modifizierer `private`



# Zugriff auf Element der Klasse A ohne Modifizierer



# Zugriff auf Element der Klasse A mit Modifizierer `protected`



# Beispiel für `private`-Zugriff (1)

```
class Alpha {
  private int iAmPrivate;
  private void privateMethod () {
    System.out.println("privateMethod");
  }
}
```

```
class Beta {
  void accessMethod() {
    Alpha a = new Alpha();
    a.iAmPrivate = 10;        // unzulässig
    a.privateMethod();        // unzulässig
  }
}
```

```
class Gamma {
  private int geheim;
  boolean gleich(Gamma anderesGamma) {
    return (this.geheim == anderesGamma.geheim); // zulässig
  }
}
```

# Beispiel für `private`-Zugriff (2)

```
class Alpha {
  private int iAmPrivate;
  private void privateMethod () {
    System.out.println("privateMethod");
  }
}
```

```
class Delta extends Alpha {
  void accessMethod(Alpha a, Delta d) {
    a.iAmPrivate = 10;    // unzulässig
    d.iAmPrivate = 10;    // unzulässig
    iAmPrivate   = 10;    // unzulässig
    privateMethod();      // unzulässig
  }
}
```

# Beispiel für `protected`-Zugriff (1)

```
class Alpha {
  protected int iAmProtected;
  protected void protectedMethod () {
    System.out.println("protectedMethod");
  }
}
```

```
class Delta extends Alpha {
  void accessMethod(Alpha a, Delta d) {
    a.iAmProtected = 10;  // zulässig
    d.iAmProtected = 10;  // zulässig
    iAmProtected   = 10;  // zulässig
    protectedMethod();    // zulässig
  }
}
```

# Beispiel für `protected`-Zugriff (2)

```
package greek;
public class Alpha {
  protected int iAmProtected;
  protected void protectedMethod() {
    System.out.println("protectedMethod");
  }
}
```

Weitere Klasse in gleichem Package:

```
package greek;
class Gamma {
  void accessMethod() {
    Alpha a = new Alpha();
    a.iAmProtected = 10;    // zulässig
    a.protectedMethod();    // zulässig
  }
}
```

# Beispiel für `protected`-Zugriff (3)

```
package greek;
public class Alpha {
  protected int iAmProtected;
  protected void protectedMethod() {
    System.out.println("protectedMethod");
  }
}
```

Unterklasse in anderem Package:

```
import greek.*;
package latin;
class Delta extends Alpha {
  void accessMethod(Alpha a, Delta d) {
    a.iAmProtected = 10; // unzulässig
    d.iAmProtected = 10; // zulässig
    iAmProtected = 10;   // zulässig
    a.protectedMethod(); // unzulässig
    d.protectedMethod(); // zulässig
    protectedMethod();   // zulässig
  }
}
```

# 9.3 Finale Deklarationen

- Der Modifizierer **`final`** kann bei Variablen, Methoden und Klassen eingesetzt werden.
- Finale Variablen (Konstanten) sind Variablen, deren Wert nach der Initialisierung nicht mehr verändert werden darf.

```
class Numbers {
  final double MAX_VALUE = 123;
  final double MIN_VALUE = -123;
}
```

- Finale Methoden sind Methoden, die nicht mehr überschrieben werden können.

```
final double wurzelAusZwei() {
   return Math.sqrt(2);
}
```

- Finale Klassen sind Klassen, von denen keine Unterklassen gebildet werden können.

```
public final class KonstanteKlasse {
   ...
}
```

# 9.4 Eigenständige, innere und anonyme Klassen

- Eigenständige Klassen:
  - Deklaration: „pro Datei eine Klasse“
  - Wiederverwendbarkeit: sehr hoch
- Innere Klasse
  - Deklaration: Klasse innerhalb einer anderen Klasse
  - Wiederverwendbarkeit : i. d. R. Verwendung nur innerhalb der äußeren Klasse oder des zugehörigen Pakets
  - Varianten: statisch und nicht-statisch
- Anonyme Klasse
  - Deklaration: innere Klasse ohne „Namen“
  - Wiederverwendbarkeit: i. d. R. nur einmalige Verwendung

# Eigenständige, innere und anonyme Klassen

Eigenständige Klassen

```
class Eins {
  int x;
  ...
  void meth1 () {
    ...
  }
  ...
}
```

```
class Zwei {
  double y;
  void meth2 () {
    Eins a = new Eins();
    a.meth1();
  }
  ...
}
```

Innere Klasse

```
class EinsMitInnererKlasse {
  int x;
  Drei d = new Drei();
  ...
  void meth1 () {
    ...
  }
  ...
  class Drei {
    double w;
    void meth3 () {
      x = 555;
    }
  }
}
```

Anonyme Klasse

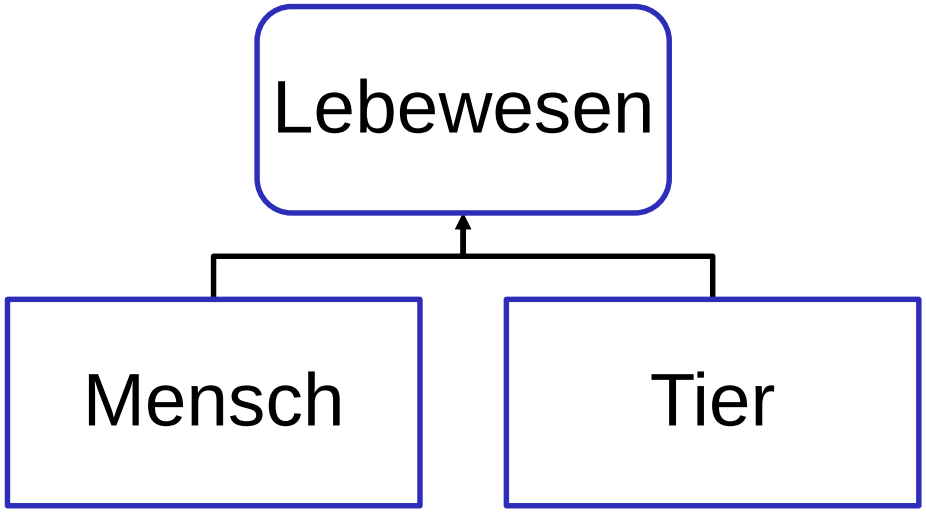
```
class EinsMitAnonymerKlasse {
  int x;
  ...
  void meth1 () {
    ...
  }
  ...
  Zwei v = new Zwei() {
    int q = 555;
    public void meth3 () {
      x = q++;
    }
  };
}
```

Name der Superklasse oder des Superinterface

# 9.5 Abstrakte Klassen

- Abstrakte Klasse:
  - Besitzt Verhalten, die sie zwar „besitzt“ aber nicht „implementiert“

- Beispiel:
  - **`Lebewesen`** sei eine abstrakte Klasse.
  - Beispiele für abstraktes Verhalten wären:
    - **`essen`**: jedes Lebewesen muss Nahrung aufnehmen
    - **`schlafen`**: jedes Lebewesen muss sich ausruhen
    - …
  - Mensch und Tier sind „konkrete“ Realisierungen



# Abstrakte Klassen

- ***Unvollständig*** implementierte Klassen als ***Rahmenvorgabe*** für Unterklassen
- Keine Objekt-Erzeugung möglich!
- Darin enthaltene Methoden können ***abstrakt*** (ohne Rumpf) sein
- Beispiel:

```
abstract class Figur {
  String name;
  Punkt ort = new Punkt();
  Figur (String name) {  // Konstruktor
    this.name = name;
  }
  abstract void show ();
     // zeigt die Daten der Figur
     // kein Methodenrumpf!
  abstract boolean contains (int x, int y);
     // prüft ob der Punkt (x,y) innerhalb der Figur liegt
     // kein Methodenrumpf!
}
```

- Konstruktoren und **`static`**-, **`final`**- oder **`private`**-Methoden können nicht abstrakt sein!

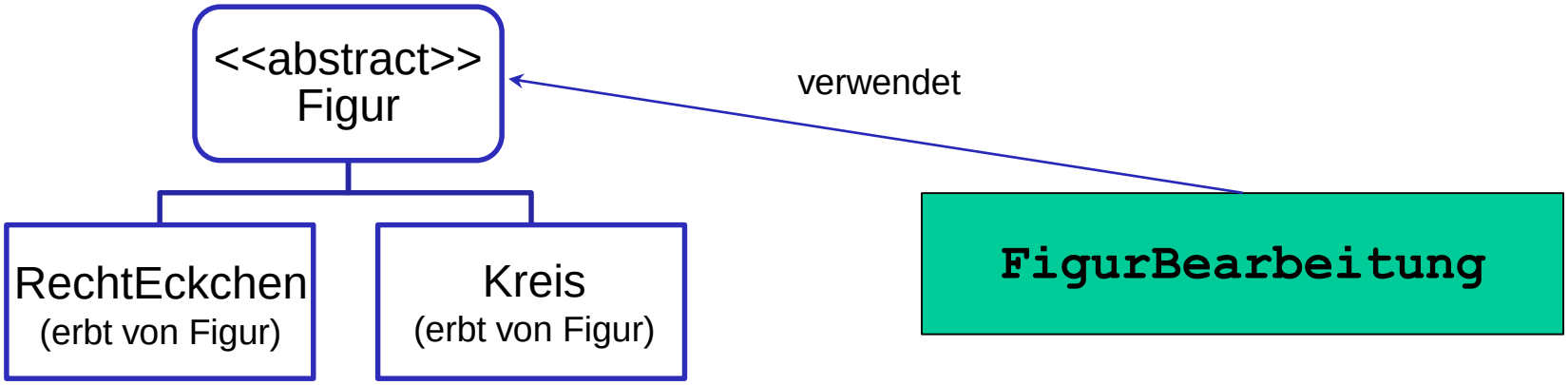
# Abstrakte Klassen

- Abstrakte Klassen können bereits compiliert und verwendet werden, lassen aber einige Details der tatsächlichen Implementierung noch offen.
- Beispiel

```
class FigurBearbeitung {
  static void check (Figur f) {
    f.show();
    if (f.contains(1,2))
      System.out.println("Punkt (1,2) liegt drin.");
  }
  ...
}
```

- Objekte vom Typ **`Figur`** können nur als Objekte von (nicht-abstrakten) Unterklassen erzeugt werden, die alle in der Klasse **`Figur`** abstrakt vorgegebenen Methoden ***implementieren***.
- Unterklassen von abstrakten Klassen können bzw. müssen aber selbst wieder abstrakt sein, nämlich dann, wenn sie selbst auch abstrakte Methoden enthalten.

# Abstrakte Klassen




**`FigurBearbeitung.check(Figur f)`**

- kennt nur Eigenschaften von „Figur“
- Interpreter sorgt dafür, dass passende Methode der Subklassen aufgerufen wird

# Abstrakte Klassen

```
abstract class Figur {
  String name;
  Punkt ort = new Punkt();
  Figur (String name) {
    this.name = name
  }
  abstract void show ();
  abstract boolean contains (int x, int y);
}
```

```
class Kreis extends Figur {
  int radius;
  Kreis (int r, int x, int y) {
    super("Kreis");      // Bezug auf Konstruktor der Oberklasse
    radius=r; ort.x=x; ort.y=y; // Variable ort geerbt
  }
  void show () {
    System.out.println(name + " mit Radius " + radius);
  }
  boolean contains (int x, int y) {
    return (ort.x-x)*(ort.x-x)+
           (ort.y-y)*(ort.y-y) <=  radius*radius;
  }
}
```

# Abstrakte Klassen

```
abstract class Figur {
  String name;
  Punkt ort = new Punkt();
  Figur (String name) {
    this.name = name
  }
  abstract void show ();
  abstract boolean contains (int x, int y);
}
```

```
class RechtEckchen extends Figur {
  int b, h;
  RechtEckchen (int x, int y, int b, int h) {
    super ("RechtEckchen");
    ort.x=x; ort.y=y; this.b=b; this.h=h;
  }
  void show () {
    System.out.println(name + " mit Breite " + b +
                               " und Hoehe " + h);
  }
  boolean contains (int x, int y) {
    return ort.x <= x && x <= ort.x+b &&
           ort.y <= y && y <= ort.y+h;
  }
}
```

# Abstrakte Klassen

- Im Beispielprogramm eingesetzt:

```
class FigurBearbeitung {
  static void check (Figur f) {
    f.show();
    if (f.contains(1,2))
      System.out.println("Punkt (1,2) liegt drin.");
  }

  public static void main(String[] args) {
    Figur f = new Kreis(5,0,0);
    check(f);
    f = new RechtEckchen(10,10,6,17);
    check(f);
  }
}
```

Ausgaben:

```
Kreis mit Radius 5
Punkt (1,2) liegt drin.
RechtEckchen mit Breite 6 und Hoehe 17
```

# Abstrakte Klassen

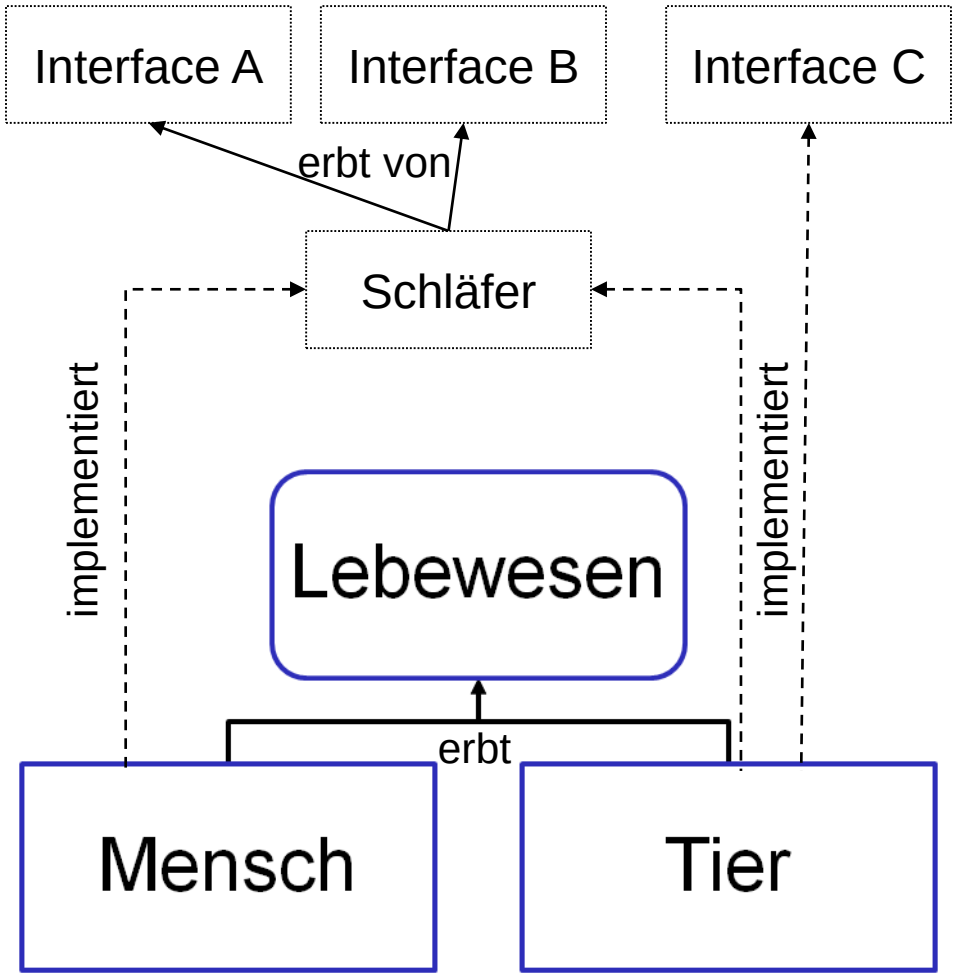
- Weiteres Anwendungsbeispiel:

```
int[] x, y;

...
Figur[] f = new Figur [5];

f[0] = new Kreis(5,10,10);
f[1] = new RechtEckchen(10,10,6,17);
f[2] = new RechtEckchen(20,10,30,7);
f[3] = new Kreis(5,10,10);
f[4] = new RechtEckchen(5,30,16,32);

...

for (int i=0;i < f.length; i++)
  if (f[i].contains(x[i],y[i]))
    f[i].show();
```

- Bei Ausführung von **`contains`** bzw. **`show`** wird stets die richtige Methode aufgerufen *(Polymorphie)!*

# 9.6 Schnittstellen (Interfaces)

- Realisierung einer „Sicht“ auf ein bestimmtes Objekt.
- Beispiel:
  - **`Schläfer`**:
    Objekte, die diese Schnittstelle „unterstützen“, können schlafen.
  - kann für **`Mensch`** und **`Tier`** gelten
  - Mehrfachvererbung in Java nicht möglich!
    - **`Mensch`** und **`Tier`** erben bereits von **`Lebewesen`**!
    - Lösung: Interfaces



# Schnittstellen (Interfaces)

- Zur Festlegung *gemeinsamer Schnittstellen* (Rahmenvorgaben) für Klassen
- Keine Methoden-Implementierungen möglich
- Ausschließlich Konstanten und abstrakte Methoden
- ***alle Methoden*** des Interface sind implizit ***abstrakt und öffentlich***
- ***alle Variablen*** des Interface sind implizit ***öffentlich, statisch und final***
- Beispiel:

```
public interface Sleeper {
  void wakeUp();       // kein Methodenrumpf
  long ONE_SECOND = 1000;    // in Millisekunden
  long ONE_MINUTE = 60000;   // in Millisekunden
}
```

# Schnittstellen (Interfaces)

- Klassen können Interfaces implementieren
- Beispiele:

```
class Circle extends Kreis implements Sleeper {
  ...
  public void wakeUp() {
    for (long i=1; i<=ONE_SECOND; i++)
      System.out.println("Chhrrrzzzzz...");
    System.out.println("Jetzt bin ich wach!");
    show();
  }
}

class Student extends Mensch implements Sleeper {
  public void wakeUp () {
    . . .
  }
}
```

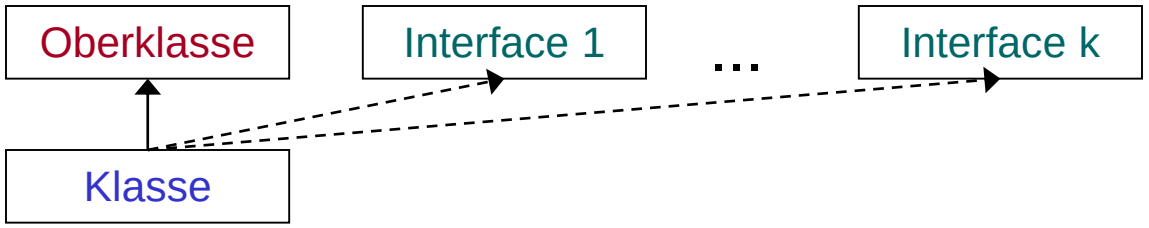
# Schnittstellen (Interfaces)

- Im Beispielprogramm eingesetzt:

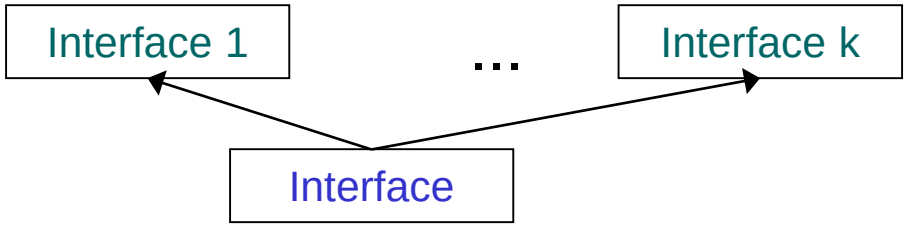
```
class SleeperTests {
  static void check (Sleeper s) {
    s.wakeUp();
  }
  public static void main(String[] args) {
    Sleeper s = new Circle(5,0,0);
    check(s);
    IOTools.readLine("ENTER-Taste betaetigen");
    s = new Student();
    check(s);
  }
}
```

# Schnittstellen (Interfaces)

- Beim Implementieren eines Interface müssen alle Methoden des Interface (auch geerbte) implementiert werden. Ausnahme: Die implementierende Klasse ist abstrakt.
- Interfaces erlauben einen gewissen Grad der *Mehrfachvererbung:*
  - jede Klasse hat maximal eine direkte Oberklasse, aber kann mehrere Interfaces implementieren.




  - jedes Interface darf mehrere Superinterfaces haben



# Schnittstellen (Interfaces)

```
interface A {
  void g();
}
```

```
interface B {
  void h();
}
```

```
interface C extends A, B {
  void f();
}
```

```
class X {
  public void h() {
     . . .
  }
  public void i() {
     . . .
  }
}
```

```
interface D {
  void i();
}
```

```
class Y extends X implements C, D {
  public void f() {
     . . .
  }
  public void g() {
     . . .
  }
}
```

Die Klasse **`Y`** muss **`f`**, **`g`**, **`h`** und **`i`** implementieren.

Die Implementierung von **`h`** und **`i`** erbt sie von der Klasse **`X`**.