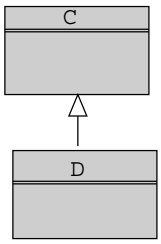
Variablen und Methoden

```
class C {
  int i = 0;
  int foo() {return  0;}
}
class D extends C {
  int i = 1;
  int foo(){return 1;}
}

public class TestCD {
public static void main(String[] args ) {
  C x = new C();
  D y = new D();
    x = y;
  System.out.println("x.i = " + x.i + " x.foo() ="
  + x.foo());
 }}
```



Ausgabe: x.i = 0 x.foo =1

Zugriff auf Attribute und Methoden

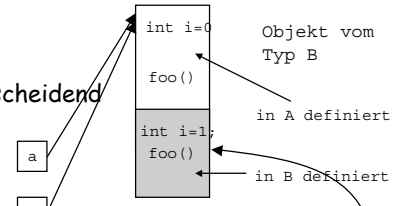
Zugriffsregel:

- für Methoden ist der Typ des Objekts entscheidend
- für Felder der Referenztyp (statisch)

Grund dafür:
Effizienter Attributzugriff wünschenswert / zwingend
- schon beim Übersetzen steht die Speicheradresse fest
- Adressen der Methoden werden sowieso dynamisch ermittelt

Regeln können explizit umgangen werden
(super, this ---> Sprachspezifikation)

Alle objektorientierten Sprachen haben ähnliche Regeln, im Detail unterscheiden sie sich meist. Vorsicht und genau hinsehen!

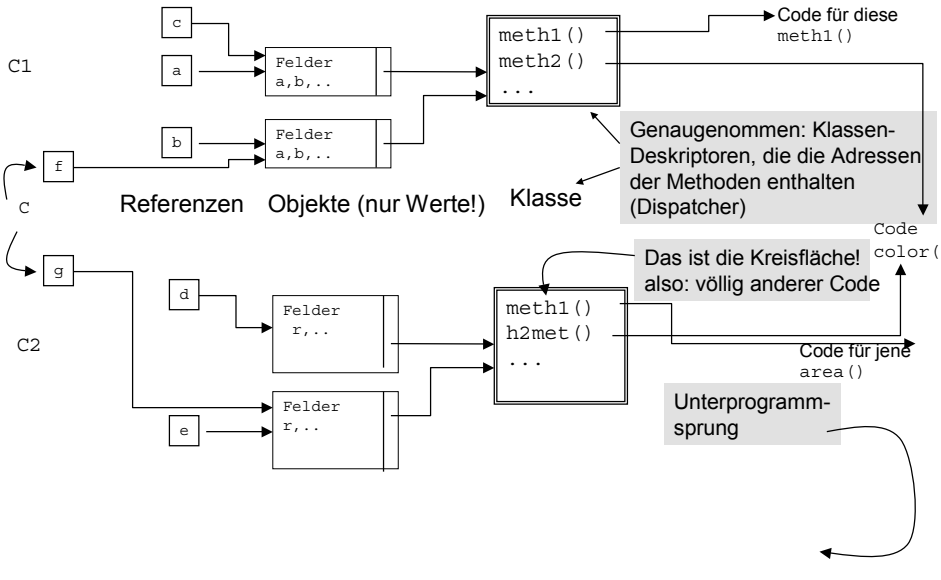



```
B extends A

B b = new b;
A a = b;
```

Totale Verwirrung?? verständlich! `equal != == ` !!.

```
a.i == 0
b.i == 1
a   == b
a.foo() führt aus
b.foo() führt aus

b.(super)foo(x)
```

Dynamischer Methodenaufruf



Spezifikation, Klassen, Schnittstellen in Java

- Schnittellen (interface)
  - trennen Spezifikation und Implementierung

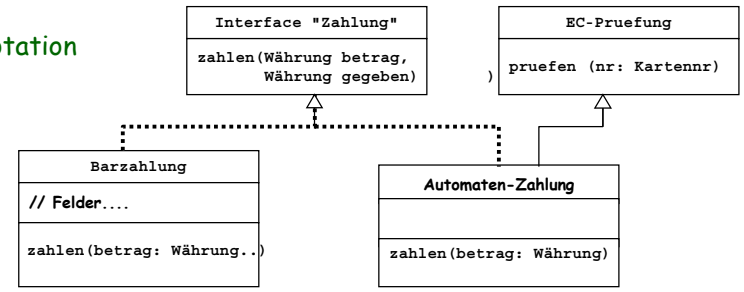
    Konsequenterweise: alle Methoden und Variablen in Schnittstelle sind **public**
  - Mehrere Implementierungen derselben Schnittstelle möglich

```
interface Stack  {// model:   [aType]
int N = 20;          // inv:  length < N = 20
void init();         // requires:  True
                     // effect:    returns []
...}

class MyStack   implements Stack {void init(){..}...}
class YourStack implements Stack {void init(){..}...}
```

Spezifikation, Klassen, Schnittstellen in Java

- .....Schnittstelle (`interface`)
  - Ermöglichen Mehrfachvererbung (-> ALP2)

UML-Notation



Spezifikation und Vererbung

```
interface Payment {// model: ...  // inv..
    Currency pay (Currency amount, Currency given);
    // requires:...  effect:...  }
  public class ECCheck {//model ...
  ... boolean  valid(int cardNr);
  }
  public class Cashpayment implements Payment {
  ...
  void pay (Currency amount, Currency given){..}
}
public class ECPayment
  extends ECCheck implements Payment {
  // model:...
     ...
  Currency pay (Currency amount, Currency given){..}
}
```

Wie sehen Modell, Invariante, Zusicherungen im Erbschaftsfall aus?

Liberalisierung der Voraussetzungen

```
interface Payment {
    // model: integers, +, -,...
    // inv total = cash + EC_total && cash > 0
    //                              && EC_total >=0
  Currency pay (Currency amount, Currency given);
    // requires: amount > 0 && given-amount >= 0
    // effects: total=TOTAL + amount
                returns given-amount
    //         TOTAL is value before payment
 }
public class ECPayment
   extends ECCheck implements Payment {
   // model:...
  Currency pay (currency amount, Currency given) {
  // Differenzbetrag given - amount soll ausge-
  // zahlt werden können ("card cashing")
```


Liberalisierung der Voraussetzung

```
void pay (Currency amount, Currency given) {
  // requires: amount >= 0 && given-amount >=0
  // effects:  cash = CASH-(given-amount) &&
  //     EC_total = EC_TOTAL + given
  //     returns amount-given
....}
```

Vererbungsregel für Voraussetzung (precondition):

Voraussetzung der Oberklassenmethode / Interface-Methode kann abgeschwächt werden

precondition (super) => precondition (sub)

Begründung:

Aufrufender der Methode muss Voraussetzung sichern.

Geschieht mit dieser Regel automatisch auch für Unterklasse / Implementierung des Interface. Beachte polymorphe Zuweisung!

Verschärfung der Effekte (Nachbedingung)

```
void pay (currency amount, Currency given){
  // requires: amount >= 0 && given-amount >=0
   // effects:  cash = CASH-(given-amount) &&
   //     EC_total = EC_TOTAL + given
   //     returns amount-given
....}
```

Regel für Effekte (Postcondition Rule):

Precondition (Super) && postcondition (sub)
=> postcondition (super)

Im Beispiel:
cash+EC_total = CASH-(given-amount) + EC_TOTAL + given = TOTAL + amount
Nachbedingung von `pay()` im Interface

Spezifikation und Vererbung

```
interface S {// model: a,b,..
              // inv:   I(a,b,..)
...op(...) throws ...
              // requires:   P(a,b,..)
              // effects:  Q(a,b,..)
}

interface I extends S {
             // model: x,y,..
  a,b,..
             // inv:   J(a,b,..,x,y,..)
...p(...)  throws ...
             // requires:   ..a,b,..,x,y,..
             // effects:  ..a,b,..,x,y,..
...op(...) throws ...
        // requires:   PP(a,b,..,x,y,..)
        // effects:  QQ(a,b,..,x,y,..)
}
```

Bedeutung:

zusätzlich zu

und `I(a,b,..)`

zusätzlich zu `op`

ersetzt altes `op`
oder `P(a,b,..)`
und `Q(a,b,..)`

(oder einfach Ersetzung der ererbten Spezifikation)