Kapitel 6

Überladungen

in C++ und Java können Funktionen überladen werden:

- in einer Klasse werden mehrere Methoden desselben Namens, aber mit unterschiedlicher Signatur definiert
- beim Aufruf wird anhand der Typen der Parameter die richtige Variante bestimmt

  Ergebnistyp spielt hingegen keine Rolle
- häufigste Anwendung: Überladung von Konstruktorfunktionen
- in C++ kann man sogar Operatoren überladen (zB + für Matrixobjekte; * für „smart Pointers“)

Achtung: Interferenz mit Vererbung!

- C++: erst Static Lookup ohne Signaturinformation, dann Überladungsauflösung innerhalb der gefundenen Klasse

  Grund: ohnehin genug Chaos wg. Konvertierungsregeln

- Java: Static Lookup bezieht von vornherein Signaturinformation mit ein $lookup(C, m, \sigma) = \max\{x \in Defs(C, m) \mid \text{type}(x) = \sigma\}$ ⇒ Signatur ist quasi Teil des Namens

- zusätzliche Probleme durch implizite Upcasts oder automatische Konversion $int \rightarrow double$ u.ä.

  falls es zusätzliche mögliche Varianten wg. Upcasting/ Konversionen gibt, wird immer die ausgewählt, die die *wenigsten Casts* erfordert - sofern dies Kriterium eindeutig ist!

  Implizite Up-Casts von this-pointern zählen in Java (JDK 1.4.2) nicht mit!!

Beispiel:

```
class C {
    public int x;
    public C(){
      x = 0;
    }
    public void show(int i){
      System.out.println("Class\t\t int");
    }
    public void show(double j){
      System.out.println("Class\t\t double");
    }
}
```

```
class SC extends C {
    public int y;
    public SC(){
      y = 1;
    }
    public void show(double j){
      System.out.println("SubClass\t double");
    }
}

class Main {
   public static void main(String args[]) {
     C c1 = new C();
     SC c2 = new SC();
     c1.show(17);        // C::show\int
     c1.show(1.0);       // C::show\double
     c2.show(19);        // C++: static lookup
                         //      -> SC::show\double
                         //      automatische
                           Konvertierung
                         //      19->double !
                         // Java: C::show\int !
     c2.show(1.0);       // SC::show\double
     ((C)c2).show(1);    // C::show\int
     ((C)c2).show(1.0); // static lookup = C::show\double
       ;
                         // dynamische Bindung -> SC::show
                           \double
   }
}
```

6.1 Überladung und dynamische Bindung

Überladung kann mit dynamischer Bindung interferieren, insbesondere auch bei Änderungen

Beispiel:

Basismodul:

```
public class S {}
public class T extends S {}
public class Base {
   public void test(S s) {
      System.out.println("base.test(S)");
   }
   public void test(T t) {
      System.out.println("base.test(T)");
   }
}
```

Modifiziertes Basismodul:

```
public class S {}
public class T extends S {}
public class Base {
   public void test(S s) {
      System.out.println("base.test(S)");
   }
}
```

Erbmodul:

```
public class Sub extends Base {
   public void test(T t) {
      System.out.println("sub.test(T)");
   }
}
public class Tester {
   public static void main(String[] args) {
      Base tester = new Sub();
      tester.test(new T());
   }
}
```

Originalprogramm: Ausgabe `sub.test(T)`

Revision: Ausgabe `base.test(S)` !!!

Grund: Entfernen der Überladung in revidierten Klasse `Base` führt dazu, dass Static lookup wg. `Base tester` den Aufruf `test` an `Base.test` bindet

Dies ist wg. $T \leq S$ typkorrekt; wg. „Signatur=Teil des Namens“ wird `test` aber in `Sub` nicht mehr redefiniert
$\Rightarrow$ keine dynamische Bindung in der Revision !!?

6.2 Smart Pointers

In C++ kann man jeden Operator überladen, zB + für Matrizen und Tensoren, ...

in C++ kann man sogar Deref-operator -> und Funktionsaufruf () überladen

Anwendung: *smart pointers*. Durch Überladen von `*`,`->`, `=` kann man zB Objekte, die in Wirklichkeit in einer Datenbank gespeichert sind, so behandeln wie gewöhnliche Objekte.

Beispiel: selbstdefiniertes, überladenes `->`

Verwendung: zB `o->m()`

Implementierung von `->` muß den tatsächlich verwendeten Pointer aus `o` errechnen

Compiler fügt am Schluss gewöhnliche Dereferenzierung mit errechnetem Pointer ein

Anwendung: voll dynamische Typisierung

- Manchmal will man dynamisch den Typ eines Objektes ändern, was in streng typisierten Sprachen eigentlich nicht geht - jedenfalls nicht, wenn die Typen nicht in Vererbungsrelation stehen

  zum geänderten Typ gehören natürlich auch andere Methodenimplementierungen

- Oft kann man das Role Pattern (s.u.) verwenden, um ein Objekt und seine Rolle zu trennen

  Rolle ist variabel, Objektidentität bleibt gleich

- Wenn man eine Zerlegung in Objekt und Rolle ($\rightsquigarrow$ Role Pattern) nicht vornehmen will/kann, helfen nur „Smart Pointers“

- Beispiel: Ein Objekt soll entweder als Hund oder als Katze agieren. Hund und Katze stehen in keiner Vererbungsrelation und haben verschiedene Methodenimplementierungen. Das Objekt soll dynamisch die Rolle wechseln können („Hundemaske aufsetzen“).

  Role-Pattern macht keinen Sinn, da Hund und Katze verschiedene Arten und nicht verschiedene Rollen sind

Hier in C++ mit überladenem Dereferenzierungsoperator:

```
class Animal {
  virtual void talk() { ... }
  virtual void think() { ... }
  virtual void act() { ... }
};

class AnimalRole {
  // muss nicht Unterklasse von Animal sein!
public:
  enum {DOG, CAT};
  AnimalRole() {
    roles[DOG] = 0;
    roles[CAT] = 0;
    _role = DOG;
  }
  Animal* operator->() {
    return roles[_role];
  }
  void role(int r) {
    _role = r;
    if (roles[_role]=0) {
       switch(r) {
         case DOG: roles[_role] = new Dog(); break;
         case CAT: roles[_role] = new Cat(); break;
       }
    }
  }

protected:
  int _role;
  Animal* roles[2];
  // hier Rollen-gemeinsame Membervariablen
};
```

```
class Dog: public Animal {
public:
  void talk() { ... "wuff" ... }
  void act() { ... }};

class Cat: public Animal {
public:
  void talk() { ... "miau" ... }
  void think() { ... }};

// Verwendung:
AnimalRole x;
x.role(AnimalRole::DOG);
x->talk(); x->think(); x->act();
x.role(AnimalRole::CAT);
x->talk(); x->think(); x->act();
```

Manche Systeme nutzen Smart Pointers exzessiv aus.

Beispiel: ODMG Standardschnittstelle zu OO-Datenbanken

```
template<class T>
class Ref {
public:
  Ref<T>(const T*);          // Konstruktor
  Ref<T>(const Ref<T>&);     //    "
  T* operator*() const;      // ueberladene
    Dereferenzierung
  T* operator->() const;     //          "
  operator T*() const        // Type Cast nach T
  Ref<T>& operator=(const T*); //  Zuweisung
  Ref<T>& operator=(const Ref<T>&); // "
}
```

6.3 Function Objects

In C++ kann man sogar die Funktionsaufruf-Syntax $f(x)$ überladen

Vorteil: Simulation von Funktionen höherer Ordnung; Übergabe von „Funktionen“ an andere Funktionen

Beispiel:

```
class A {
  void operator () (int x) { ... x ...};
  A operator () (double x) { ...x ... return ...};
 ...
};
class B {
  void m(A a) {
   ...
   a(5);
   a = a(3.14);
};
```

$\rightsquigarrow$ Extraskript generische Progr. in C++

Bem.: Echte Funktionen höherer Ordnung zB in Haskell sind natürlich viel schöner!

6.4 Multimethoden

in gängigen OO-Sprachen richtet sich dynamische Bindung nur nach Typ des Bezugsobjekts. Dies ist manchmal unnatürlich.

Idee: dynamic dispatch nach Typ des Bezugsobjektes *und* der Parameter

Konsequenz: Methodenimplementierung für alle Kombinationen nötig!

bisher nur von experimentellen Sprachen angeboten.

Vorteil: schwache Kopplung, bessere Erweiterbarkeit, bessere Struktur

abstraktes Szenario: MultiJava

```
class A {
  f(A x) { }

  f(B x) { }
}

class B extends A {
  f(A x) { }

  f(B x) { }
}

o1 = new A|B; o2 = new A|B;

o1.f(o2);
```

„double dispatch“: zur *Laufzeit* gibt es 4 mögliche Ziele des Methodenaufrufs!

Achtung: Multimethoden-Deklarationen sehen aus wie Überladungen, sind aber ein dynamischer Mechanismus!

Anwendungsbeispiel: binäre Operationen in Verbindung mit Subtyping

```
interface NotRational {
  NotRational plus(NotRational x);
  NotRational equals (NotRational x);
}

class Real implements NotRational {
  double value;
  Real plus (Complex x) {
    ...
  }
  Real plus (Real x) {
    ...
  }
}
class Complex {... analog...}

Real a,b,c;
c = a.plus(b);
double d = a.plus(b).value;
NotRational x,y;
if (input(...))
  x = new Real();
else
  x = new Complex();
y.plus(x);
```

Laufzeitmechanismus:

- zu jeder Methode gibt es mehrere Implementierungen mit verschiedenen Parametertypen (“Multimethode”)
- gewöhnliche dynamische Bindung bestimmt anhand Bezugsobjekt Klasse (bzw Subobjekt), aus der Methodendefinition kommen muss
- implizite Fallunterscheidung anhand des dynamischen Parametertyps wählt Methoden-Variante

Vorsicht: kann zu Verletzungen der Typ-Kontravarianz führen; Obacht mit Laufzeitfehlern

Versuch der Simulation in Standard Java:

Versuch 1:

```
class Real implements NotRational {
  double value;
  NotRational plus (NotRational x) {...}
}
class Complex implements NotRational {
   NotRational plus (NotRational x) {...}
}
Real a,b; double c = ((Real) a.plus(b)).value;
```

⇒ unsicherer Downcast erforderlich

Versuch 2: Methodenüberladung nebst Fallunterscheidung

```
class Real implements NotRational {
  double value;
  NotRational plus (NotRational x) {
    if (x instanceof Real) {...Code A...}
    else if (x instanceof Complex ) {...Code B...}
  }
  Real plus (Real x) {...Code C...}
}
Real a,b,c;
c = a.plus(b);
double d = a.plus(b).value;
```

⇒ Code A muss gleich Code C sein: Coderedundanz; ferner explizite Fallunterscheidung. Zwei softwaretechnische Todsünden!

Multimethoden können softwaretechnisch sinnvoll sein!