Objektorientierte Programmierung II: OO in Modula-3

- **Objekttypen für Klassen**
- **Untertypen für opake Klassen**
- **Vererbung zwischen opaken Klassen**
- **Diskussion**

Wiederholung ADT

- **ADTs in Modula-3**
  - ein ADT ist immer ein **Referenztyp**
  - in der Schnittstelle wird ein **opaker Typ als Untertyp** des vordefinierten Referenztyps REFANY deklariert
- **Ein ADT entspricht dem Konzept einer Klasse.**
- **Eine Exemplar eines ADTs entspricht einem abstrakten Objekt**
- **Mit Hilfe der ADTs**
  - kann eine **klassenbasierte** Programmierung in Modula-3 umgesetzt werden.
- **Zur Objektorientierung fehlen noch**
  - Sprachkonstrukte, um die **Vererbung** zu modellieren.
- **Hierfür**
  - stellt Modula-3 das Konzept der **Objekttypen** bereit.

Objekttypen für Klassen

Objekttypen in Modula-3

- **Mit Hilfe der Objekttypen können in Modula-3 Klassen beschrieben werden.**
- **Ein Objekttyp gibt an**
  - Bezeichner der Klasse (des Objekttyps)
  - Bezeichner der Oberklasse
    - Es kann maximal eine Oberklasse geben

    Einfachvererbung
  - Bezeichner der Exemplarvariablen
  - Signaturen der Methoden
  - Bezeichner der Methoden, die in der Klasse redefiniert werden
    - in Modula-3 spricht man von überschreiben (overrides)
- **Ein Objekttyp (Klasse) wird**
  - in einer Modulschnittstelle deklariert
  - die Implementierung enthält den strukturellen Aufbau des Objekttyps (Klasse) und die Realisierung der Methoden.

Objekttypen für Klassen

Beispiel Objekttyp : Schnittstelle

INTERFACE Point

```
TYPE Point = ROOT OBJECT
                x : INTEGER; y : INTEGER;
             METHODS
                getX(): INTEGER          := PointGetX;
                setX (value : INTEGER)   := PointSetX;
                getY (): INTEGER         := PointGetY;
                setY (value : INTEGER)   := PointSetY;
                add (p: Point) : Point   := PointAdd;
                . . .
...          END;
```

- **Die Klasse**
  - hat die vordefinierte Klasse **ROOT** als Oberklasse
  - deklariert zwei Exemplarvariablen x und y
- **In der METHODS-Klausel**
  - kann die **Bindung** zwischen Methode und der Prozedur, die sie realisiert, hergestellt werden!
- **Objekttypen sind spezielle Referenztypen**
  - Objekte werden mit der **NEW**-Operation erzeugt.

Objekttypen für Klassen

Beispiel Objekttyp: Rumpf des Moduls, Verwendung der Klasse

Die Prozeduren, die Methoden realisieren, erwarten als ersten Parameter ein Objekt der Klasse.

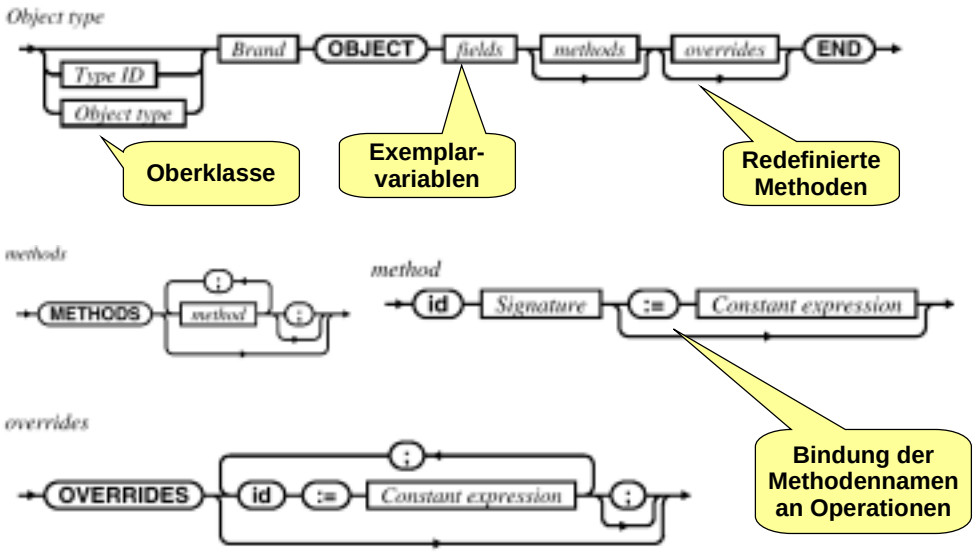
```
MODULE Points;

PROCEDURE PointGetX(self : Point): INTEGER =
BEGIN
  RETURN self.x;
END PointGetX;

PROCEDURE PointSetX(self: Point; value : INTEGER) =
BEGIN
  self.x := value;
END PointSetX;
...

PROCEDURE PointAdd (self: Point; p: Point) : Point =
VAR newP : Point;
BEGIN
  newP := NEW(Point);
  newP.setX(self.x + p.getX());
  newP.setY(self.y + p.getY());
  RETURN newP;
END PointAdd;
...
```

Verwendung der Klasse `Point`

```
VAR p1, p2, p3 : Point;
BEGIN
  p1 := NEW(Point);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
END Point_Test.
```


Objekttypen für Klassen

Syntax der Objekttyp-Deklaration



Objekttypen für Klassen

Operationenbindung und Redefinition

```
TYPE Geo_Object =  ROOT OBJECT
       METHODS
        moveTo (p : Point):= NIL ;
        contains (p : Point) : BOOLEAN:= NIL;
       END;
```

abstrakte Klasse

```
TYPE Rectangle = Geo_Object OBJECT
         origin : Point;
         corner : Point;
       METHODS
         area() : INTEGER      := RectArea;
         height () : INTEGER   := RectHeight;
         setOrigin (p : Point) := RectOrigin;
         getOrigin(): Point    := RectGetOrigin;
         …
         asText(): TEXT        := RectAsText;
       OVERRIDES
         moveTo   := RectMoveTo;
         contains := RectContains;
       END;
```

konkrete Klasse

konkrete Klasse

```
TYPE DispRectangle = Rectangle OBJECT
         color : TEXT;
       METHODS
         setColor(c : TEXT) := DispRectSetColor;
         getColor() : TEXT  := DispRectGetColor;
         draw(): TEXT       := DispRectDraw;
       OVERRIDES
         asText := DispRectAsText
       END;
```


Untertypen für opake Klassen

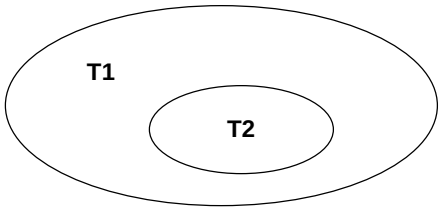
Opake Objekttypen

- **Um opake (geschützte) Objekttypen zu deklarieren**
  - verwenden wir die M3-Sprachkonstrukte, die wir zur Realisierung ADTen benutzt haben!
    - **Trennung** von Schnittstelle und Implementierung
      - Pro Objekttp (Klasse) ein Modul
    - **Opake** Typen
    - Konzept der **Untertypen**
    - Konvention der **T-Notation** für Objekttypen
- **Der Name der Klasse wird als opaker Typ deklariert**
  - Deklaration in der Schnittstelle.
  - Ein opaker Typ muß immer ein Untertyp eines Referenz- oder Objekttyps sein.
  - Der Obertyp (ein Objektyp) deklariert die **öffentliche** Schnittstelle
    - D.h. nur die Signaturen der Methoden werden angegeben

Untertypen für opake Klassen

Untertypen in Modula-3

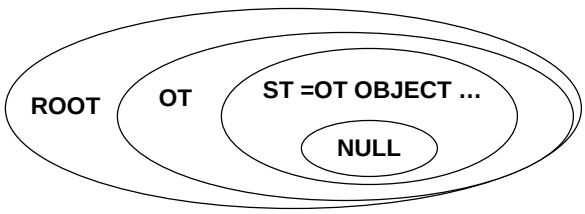
- **Subtypen in Modula-3**
  - Modula-3 kennt das Konzept der Konstruktion von ***Untertypen***
  - Subtypbeziehung wird durch "***<:***" angezeigt
- **Definition**
  - Seien T1 und T2 Typen und die Relation T2 <: T1 besteht, dann sind ***alle Werte*** von T2 auch ***Werte*** von T1
  - T1 nennt man ***Obertyp***; T2 nennt man ***Untertyp***
  - Es gilt: ein Typ kann beliebig viele Untertypen haben, ein Typ kann ***maximal*** einen Obertyp haben.



Untertypen für opake Klassen

Untertypen von Referenz- und Objekttypen

- **Modula-3 definiert zwei vorgegebene Referenztypen**
  - ***REFANY*** und ***NULL*** mit folgender Subtypbeziehung
  - `NULL <: REF T <: REFANY`
- **Modula-3 definiert einen vorgegebenen Objekttyp**
  - ROOT mit folgender Subtypbeziehung
  - `NULL <: ST = OT OBJECT … END <: OT <: ROOT <: REFANY`
- **Interpretation:**
  - jeder Objekttyp ist ***Untertyp*** von ROOT (und damit ein Referenztyp)



Untertypen für opake Klassen

Beispiel: opaker Objekttyp, Schnittstelle

```
INTERFACE Point;

TYPE Point <: PublicPoint;

     PublicPoint = ROOT OBJECT
             METHODS
               getX(): INTEGER;
               setX (value : INTEGER);
               getY (): INTEGER;
               setY (value : INTEGER);
               add (p: Point) : Point;
               minus (p : Point) : Point;
               asText(): TEXT;
             END;
END Point.
```

Point ist Untertyp des "öffentlichen Teils von Point"

Dieser Typ wird exportiert und kann benutzt werden!!!!!!

In der Implementierung müssen die Exemplar-variablen und die Operationen, die die Methoden realisieren, angegeben werden!

Point <: PublicPoint <: ROOT

Untertypen für opake Klassen

Opaker Objekttyp: Rumpf

```
MODULE Point;

REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX;
        getX := PointGetX;
        setY := PointSetY;
        getY := PointGetY;
        add := PointAdd;
        minus := PointMinus;
        asText := PointAsText;
      END;

PROCEDURE PointGetX(self : Point): INTEGER
=
BEGIN
  RETURN self.x;
END PointGetX;
...
END Point.
```

Rumpf der Klasse

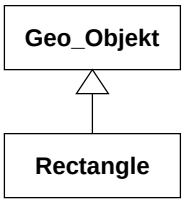
- **Point wird als Objekttyp deklariert**
  - PublicPoint ist der **Obertyp**
  - Point <: PublicPoint <: ROOT
- **Point erweitert den Obertyp**
  - um die fehlenden **Instanzvariablen.**
- **Point bindet**
  - an die Methoden entsprechende **Implementierungen.**

Untertypen für opake Klassen

Diskussion dieser Realisierung

- **Untertyp-Konzept wird verwendet, um**
  - **Spezialisierungsbeziehung** zwischen (hier offenen) Klassen auszudrücken

```
TYPE Rectangle = Geo_Object OBJECT
         origin : Point;
         corner : Point;
       METHODS
         area() : INTEGER      := RectArea;
         …
         asText(): TEXT        := RectAsText;
       OVERRIDES
         moveTo   := RectMoveTo;
         contains := RectContains;
       END;
```



  - geschützte Objekttypen (Klassen) zu **implementieren**

```
REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX; ...
      END;
```


Vererbung zwischen opaken Klassen

T-Konvention

T bezeichnet immer den im Modul realisierten Objekttyp (Klasse)

Verwender

```
MODULE Point_Test  EXPORTS Main;

IMPORT Point,  SIO;

VAR p1, p2, p3 : Point.T;
BEGIN
  p1 := NEW(Point.T);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point.T);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
  SIO.PutLine(p3.asText());
END Point_Test.
```

Schnittstelle

```
INTERFACE Point;

TYPE T <: Public;

          Public = ROOT OBJECT
              METHODS
                setX(): INTEGER;
                ...
                add (p: T) : T;
                minus (p : T) : T;
                asText(): TEXT;
              END;
END Point.
```

Rumpf

```
MODULE Point;

  REVEAL T = Public BRANDED OBJECT
        x:INTEGER;
        y:INTEGER;
     OVERRIDES
        setX:= PointSetX;
        ...
     END;
  ...
```

Vererbung zwischen opaken Klassen

Implementierung der Klasse Point

```
PROCEDURE AsText (self : T) : TEXT =
BEGIN
  RETURN (Fmt.Int(self.x)  &  "&"  &  Fmt.Int(self.y));
END AsText;

PROCEDURE GetX(self : T): INTEGER =
BEGIN
  RETURN self.x;
END GetX;

PROCEDURE GetY(self : T): INTEGER =
BEGIN
  RETURN self.y;
END GetY;

...

PROCEDURE Add (self: T; p: T) : T =
VAR newP : T;
BEGIN
  newP := NEW(T);
  newP.setX(self.getX() + p.getX());
  newP.setY(self.getY() + p.getY());
  RETURN newP;
END Add;
```


Vererbung zwischen opaken Klassen

Beispiel: Klassenhierarchie Geo_Objects

```
INTERFACE Geo_Object;
IMPORT Point;

TYPE T =  ROOT OBJECT
  METHODS
    moveTo (p : Point.T);
    contains (p : Point.T) : BOOLEAN;
  END;
END Geo_Object.
```

Abstrakte Klasse

```
INTERFACE Rectangle;
IMPORT Geo_Object, Point;

TYPE T <: Public;
     Public = Geo_Object.T OBJECT
              METHODS
                area() : INTEGER;
                height () : INTEGER;
                setOrigin (p : Point.T);
                getOrigin(): Point.T;
                setCorner(p : Point.T);
                getCorner(): Point.T;
                asText () : TEXT;
              END;
END Rectangle.
```

```
INTERFACE DisplayableRectangle;
IMPORT Rectangle;

TYPE T <: Public;
     Public = Rectangle.T OBJECT
              METHODS
                setColor(c : TEXT);
                getColor() : TEXT;
                draw(): TEXT;
              END;
END DisplayableRectangle.
```

Vererbung zwischen opaken Klassen

Implementierung von Rectangle

```
MODULE Rectangle;
IMPORT Point;

REVEAL
  T = Public BRANDED OBJECT
        origin : Point.T;
        corner : Point.T;
      OVERRIDES
        setOrigin := SetOrigin;
        getOrigin := GetOrigin;
        setCorner := SetCorner;
        getCorner := GetCorner;
        area := Area;
        height := Height;

        moveTo   := MoveTo;
        contains := Contains;
        asText   := AsText;
      END;

PROCEDURE AsText(self : T) : TEXT =
BEGIN
  RETURN ("Rect origin: " & self.origin.asText() &
             " corner: " & self.corner.asText());
END AsText;
```

Notwendig, um geschützte Implementierung zu erhalten

Echte Redefinitionen

```
PROCEDURE MoveTo (self : T; p : Point.T)=
VAR newPoint := NEW(Point.T);
BEGIN
  newPoint := self.getCorner();
  self.setCorner(newPoint.add(p));

  newPoint := self.getOrigin();
  self.setOrigin(newPoint.add(p));
END MoveTo;


PROCEDURE Contains (self : T; p : Point.T):
                    BOOLEAN =
BEGIN
  ...
END Contains;
```


Vererbung zwischen opaken Klassen

Implementierung von `DispRectangle`

```
MODULE DisplayableRectangle;

IMPORT Rectangle;
TYPE SuperClass = Rectangle.T;

REVEAL
  T = Public BRANDED OBJECT
        color : TEXT;
      OVERRIDES
        setColor := SetColor;
        getColor := GetColor;
        draw     := Draw;

        asText := AsText;
END;
```

Echte Redefinition

- **Direkter Aufruf einer Methode der Oberklasse**
  - Sollte im Normalfall **nicht** gemacht werden.
  - Sinnvoll jedoch, wenn **rekursiv redefiniert** wird, d.h. daß die Leistung der redefinierten Methode bei der neuen Implementierung verwendet werden soll.

```
PROCEDURE AsText(self : T) : TEXT =
VAR t : TEXT;
BEGIN
  t := SuperClass.asText(self);
  RETURN (t & " color: " & self.color );
END AsText;
```

Verwenden der gerade redefinierten der Oberklasse

Vererbung zwischen opaken Klassen

Umgang mit Objekten

```
IMPORT Point,  SIO, Rectangle, DisplayableRectangle;
VAR
   p1, p2, p3 := NEW(Point.T);
   r1         := NEW(DisplayableRectangle.T);

BEGIN
  p1.setX(10);
  p1.setY(10);
  SIO.PutLine(p1.asText());

  p2.setX(100);
  p2.setY(100);
  SIO.PutLine(p2.asText());

  r1.setOrigin(p1);
  r1.setCorner(p2);
  r1.setColor("black");
  SIO.PutLine(r1.asText());

  p3.setX(50);
  p3.setY(50);
  SIO.PutLine(p3.asText());

  r1.moveTo(p3);
  SIO.PutLine(r1.asText());
```

Deklarieren der Variablen und erzeugen der Objekte

Senden von Nachrichten an die erzeugen Objekte

```
10&10
100&100
Rect origin: 10&10 corner: 100&100 color: black
50&50
Rect origin: 60&60 corner: 150&150 color: black
```


Diskussion

Objektorientierung in M3

- **Feststellung:**
  - Es ist **möglich**!
- **Anwendbarkeit und Verbindung mit anderen Paradigmen**
  - Modula-3 ist eine **hybride** Sprache
  - Sie erlaubt **imperative** und **objektorientierte** Programmierung
  - Beides kann bunt durcheinander **gemischt** werden
    - Vorteil: Jedes Konzept an seinem Platz
    - Nachteil: Mischung, unüberlegt angewendet, kann undurchsichtig werden
- **Die Implementierung der objektorientierten Konzepte**
  - basiert auf dem Untertyp-Konzept der Sprache
  - die OO-Konzepte können damit umgesetzt werden
- **Konsequenz**
  - Soll durchgängig objektorientiert entwickelt werden, dann sollte man eine **rein objektorientierte** Sprache verwenden, z.B. Java, Eiffel, Smalltalk!
  - die Vorteile eines hybriden Ansatzes sind nicht mehr gegeben

Was haben wir gelernt?

- Realisierung von Klassen in Modula-3
- Objekttypen zur Formulierung von Klassen in der Schnittstelle von Modulen
- offene Klassen (nicht zu empfehlen, wider Datenabstraktion), opake Klassen
- Unterkonzept für opake Klassen (Information Hiding) und für Vererbung
- Je Klasse ein Modul und eine Schnittstelle
- In Modula-3 kann objektorientiert programmiert werden (etwas umständlich)
- T-Konvention
- Vorteil hybrider Sprachen: OO und objektbasierte, adt-basierte sowie prozedurale Programmierung
- Nachteil: Programme können sehr unübersichtlich sein; reine OO-Programme auch
- Objektorientierte Programmierung schwierig: Struktur ist nur die Klassenhierachie, saubere Klassenhierachien zu modellieren, ist schwer

Glossar

- ADT bzw. Klasse, Realisierung durch ADT-Modul bzw. opake Klasse in Interface mit zugehörigem Rumpf
- Objekttypen von Modula-3, Deklaration in einer Schnittstelle
- Examplarvariable, Signatur von Methoden
- Redefinition und Binden von Prozeduren an die Methoden der Signatur bzw. der redefinierten Methoden
- offene Klassen, opake Klassen
- Schnittstelle für opake Klasse, Details im Rumpf über die REVEAL-Klausel
- Aufruf einer Methode der gleiche Klasse über `self`
- Aufruf einer Methode der Oberklasse über `super`