Abteilung für Wirtschaftsinformatik

Automatisierung von Windows Anwendungen (4)

Abstrakter Datentyp, Klassen, Methoden, Attribute, Nachrichten, Geltungsbereiche, Generalisierungshierarchie, Vererbung

Prof. Dr. Rony G. Flatscher

Wirtschaftsuniversität Wien ■ Augasse 2-6 ■ A-1090 Wien

Datentyp (DT)

- Datentyp
  - Definiert die Menge der zulässigen Werte
  - Definiert die erlaubten Operationen (z.B. Addieren, Verketten)
  - Beispiele
    - Datentyp **Geburtstag**
      - Z.B. definiert als gültiges Datum mit gültiger Zeit
      - Erlaubte Operationen, z.B.: Datum und Zeit ändern beziehungsweise abfragen
    - Datentyp **Person**
      - Z.B. definiert als Vor- und Nachname, Gehalt
      - Erlaubte Operationen, z.B.: Vor- und Nachname sowie Gehalt ändern beziehungsweise abfragen, Gehalt erhöhen

Datentyp (DT)
Classic Rexx, Probleme

- Keine Möglichkeit, *ausdrücklich* Strukturen zur Repräsentation eines Datentypes zu bilden
- Keine Möglichkeit, *ausdrücklich* Operationen festzulegen, die *ausschließlich* für einen bestimmten Datentyp gelten
- Versuch der Verkodierung mit Hilfe von
  - Zeichenketten
  - Stem-Variablen

Datentyp (DT)
Classic Rexx, mögliche Lösung 1

- Verkodierung mit Hilfe von **Zeichenketten**
  - z.B. Daten vom Typ ***Geburtstag***

    ```
    "20050901 16:00"
    "20080229 19:19"
    ```

  - z.B. Daten vom Typ ***Person***

    ```
    "Albert Einstein 45000"
    "Vera Mitirgendeinemnamen 25000"
    ```

  - Abarbeitung nur möglich mit Kenntnis von
    - **Anzahl** und **Reihenfolge** der DT-"Felder" (Spalten)
    - **Dimension** der Spalten (variabel, fix)
    - Zum Beispiel verkodierte ASCII-Dateien
      - Variable Spaltenbreite, daher Trennzeichen notwendig
        - z.B. "Comma Delimited Format"
      - Fixe Spaltenbreite

Datentyp (DT)
Classic Rexx, mögliche Lösung 2

- Verkodierung mit Hilfe von **Stems**
  - z.B. Daten vom Typ ***Geburtstag***
    - Sammlung der zeichenkettenkodierten Daten mit Hilfe von Stems

```
geb.1 = "20050901 16:00"
geb.2 = "20080229 19:19"
```

      - Abarbeitung nur möglich mit Kenntnis von **Anzahl, Reihenfolge** und **Spaltenbreite** der DT-"Felder", z.B. SysFileTree()
    - **Strukturierung** und Sammlung der zeichenkettenkodierten Daten mit Hilfe von Stems

```
geb.1.eDatum = "20050901"
geb.1.eZeit  = "16:00"
geb.2.eDatum = "20080229"
geb.2.eZeit  = "19:19"
```

      - Abarbeitung **lediglich** mit Kenntnis der Bezeichner für die einzelnen DT-"Felder" möglich !

Datentyp (DT)
Classic Rexx, mögliche Lösung 3

- Verkodierung mit Hilfe von **Stems**
  - z.B. Daten vom Typ ***Person***
    - **Strukturierung** mit Hilfe von Stems

```
pers.eVorname       = "Albert"
pers.eFamilienname  = "Einstein"
pers.eGehalt        = "45000"
```

  - beziehungsweise:

```
pers.eVorname       = "Vera"
pers.eFamilienname  = "Mitirgendeinemnamen"
pers.eGehalt        = "25000"
```

    - Bei Nutzung eines Stems **muss** ein zusätzlicher Index eingeführt werden, damit die beiden obigen Personen getrennt voneinander gespeichert werden können!
    - Die letzte Zuweisung ("Vera") würde die erste ("Albert") vollständig überschreiben

Datentyp (DT)
Classic Rexx, Diskussion der möglichen Lösungen

- Datenstrukturen von DT
  - Kodierung in Zeichenketten oder Stems
    - Krücke, da implementationsabhängig!
    - Fehleranfällig
- Erlaubte Operationen
  - Keine Möglichkeit, erlaubte Operationen zu definieren!
    - Funktionen und Prozeduren müssen für sich allein definiert werden
    - Direkter Zugriff auf Zeichenketten und auf Stems **muss** über EXPOSE-Anweisungen ermöglicht werden
      - Probleme mit Geltungsbereichen, Fehlerquellen
- Isolationsschutz ("Kapselung") der einzelnen DT-Ausprägungen ("Instanzen") nicht möglich

Abstrakter Datentyp (ADT)

- Abstrakter Datentyp
  - **Schema** zur Umsetzung von Datentypen
    - Definition von **Attributen**
      - Ergibt Datenstruktur
    - Definition von **Operationen** (“Verhalten”)
      - Funktionen, Prozeduren
  - Interne Datenstrukturen und Werte in der Regel
    - Nicht von außerhalb sichtbar
    - Nicht direkt von außerhalb manipulierbar
    - **Kapselung !**
  - **Schema** muß in einer *geeigneten* Programmiersprache umgesetzt werden
    - Classic Rexx nicht dafür geeignet
    - Object Rexx wie jede objektorientierte Sprache dafür geeignet

Abstrakter Datentyp (ADT)
Umsetzung in Object Rexx

- Abstrakter Datentyp
  - **Schema** zur Umsetzung von Datentypen
    - **::CLASS**-Direktive
      - Definition von **Attributen** und damit der internen Datenstruktur
        - **EXPOSE**-Anweisung **innerhalb** von Methoden oder
        - **::**METHOD-Direktive mit der Kennzeichnung als **ATTRIBUTE**
      - Definition von **Operationen** (Funktionen, Prozeduren)
        - ::METHOD-Direktive
  - Instanzen von Klassen ("Objekte")
    - Einzelne, voneinander eindeutig unterscheidbare Ausprägungen desselben Typs
    - Weisen alle dieselben, in der Klasse vordefinierten Attribute und Operationen auf

Abstrakter Datentyp (ADT)
Beispiel: Definition eines ADT

- Object Rexx-Umsetzung des ADT ***Geburtstag***

```
/**/
::CLASS  Geburtstag
::METHOD Datum ATTRIBUTE
::METHOD Time  ATTRIBUTE
```

- Objekt
  - Instanz (Ausprägung) eines ADT, d.h. einer Klasse
    - Eineindeutig, von anderen Objekten des gleichen Typs unterscheidbar
  - Erzeugen: Nachricht **NEW** an Klasse senden
    - Zugriff auf Klasse über die Umgebungsbezeichnung
      - Punkt unmittelbar gefolgt vom Klassenbezeichner, z.B.

```
Objekt1 = .String~NEW("hallo")
Objekt1 = "hallo"
```

Object Rexx
Nachrichten

- **Interaktion** (Aktivieren von Funktionen/Operationen) **mit Objekten** (Instanzen) **ausschließlich** über Nachrichten, die an Objekte gesendet werden
  - Namen der Nachrichten entsprechen den Bezeichnern der Methoden
  - Nachrichtenoperator ("***Twiddle***") ist das Tilde-Zeichen: **~**
    - z.B. "ABC"~REVERSE ergibt: CBA
  - "Kaskadierende" Nachrichten, zwei Twiddles: **~~**
    - z.B. "ABC"~~REVERSE ergibt (**Achtung!**): ABC
    - Gesendete Nachrichten aktivieren die entsprechenden Methoden des Objekts, Rückgabewert ist aber **immer** das Objekt selbst!
      - Somit können mehrere Nachrichten hintereinander ("kaskadierend") an ein- und dasselbe Objekt gesandt werden
  - Abarbeitung von Nachrichten: von links nach rechts

Abstrakter Datentyp (ADT)
Beispiel: Benutzung eines ADT

- Object Rexx-Umsetzung des ADT ***Geburtstag***

```
/**/
g1 = .Geburtstag~New
g1~Datum= "20050901"
g1~Time= "16:00"
g2=.Geburtstag~New~~"Datum="("20080229")~~"Time="("19:19")
SAY g1~Datum g2~Datum g1~Time g2~Time
```

```
::CLASS  Geburtstag
::METHOD Datum ATTRIBUTE
::METHOD Time  ATTRIBUTE
```

Ausgabe:

```
20050901 20080229 16:00 19:19
```

Abstrakter Datentyp (ADT)
Beispiel: Benutzung eines ADT, 2

- Object Rexx-Umsetzung des ADT ***Geburtstag***

```
/**/
g1 = .Geburtstag ~New
g1 ~Datum="20050901"
g1 ~Time = "16:00"
g2=.Geburtstag ~New ~~"Datum="("20080229") ~~"Time="("19:19")
SAY g1~Datum g2~Datum g1 ~Time g2~Time
```

```
::CLASS  Geburtstag
::METHOD Datum ATTRIBUTE
::METHOD Time  ATTRIBUTE
```

Ausgabe:

```
20050901 20080229 16:00 19:19
```

Geltungsbereich (1)

- "Geltungsbereich" auf Englisch: "Scope"
  - Synonym: “Reichweite”
  - Definiert die Sichtbarkeit von Sprungmarken, Variablen, Klassen, Routinen, Methoden und Attributen
- Vgl. auch Aufsatz

  http://wi.wu-wien.ac.at/rgf/rexx/orx07/Local.pdf
- "**Standard Scope**"
  - Legt fest, welche Sprungmarken sichtbar sind
    - Sprungmarken sind nur innerhalb eines Programmes sichtbar (bis zum Ende des Programmes **oder** zur ersten Direktive, die durch einen Doppelpunkt :: eingeleitet wird, was immer früher kommt)
    - Sprungmarken innerhalb einer ::ROUTINE- und ::METHOD-Direktive sind auch nur darin sichtbar

Geltungsbereich (2)

- "**Procedure Scope**"
  - Legt fest, welche Variablen des Aufrufers innerhalb der aufgerufenen Prozeduren/Funktionen sichtbar sind
    - Sprungmarken, **ohne** PROCEDURE-Anweisung
      - Zugriff auf sämtliche Variable des aufrufenden Programmteiles
    - Sprungmarken, gefolgt von der PROCEDURE-Anweisung
      - Zugriff auf Variable des aufrufenden Programmteiles **nicht** möglich, werden "verdeckt"
        - "**Lokaler Geltungsbereich**"
        - **Allerdings:** mit Hilfe einer EXPOSE-Anweisung, die einer PROCEDURE-Anweisung folgt, kann gezielt der Zugriff auf Variablen des aufrufenden Programmteiles ermöglicht werden

Geltungsbereich (3)

- "**Program Scope**"
  - Legt fest, daß die in einem Programm definierten Klassen und Routinen in jedem Fall zur Verfügung stehen
    - **Lokale Klassen** und **Routinen** kann man nicht überschreiben
    - Klassen und Routinen können als **öffentlich** definiert werden
  - Legt darüber hinaus fest, welche *öffentlich zugänglichen **Klassen*** und *öffentlich zugänglichen **Routinen*** aus *aufgerufenen* oder wegen der ::REQUIRES-Direktive *aufgesuchten* Programme zur Verfügung stehen
    - **Achtung!**
      - Werden **zwei** verschiedene Programme hintereinander aufgerufen, die eine **öffentliche Klasse** oder **öffentliche Routine** mit **demselben Namen** definieren, dann wird der Zugriff auf jene Klasse/Routine eingerichtet, die im **zuletzt** aufgerufenen Programm definiert wurde.

Geltungsbereich (4)

- "**Routine Scope**"
  - Bildet einen eigenständigen Geltungsbereich für
    - Sprungmarken ("Standard Scope") und
    - Variable ("Procedure Scope")
  - Der Zugriff auf Klassen und Routinen erfolgt nach den Regeln des "Program Scope"

Geltungsbereich (5)

- "**Method Scope**"
  - Bildet einen eigenständigen Geltungsbereich für
    - Sprungmarken ("Standard Scope") und
    - Variable ("Procedure Scope")
  - Der Zugriff auf Klassen und Routinen erfolgt nach den Regeln des "Program Scope"
  - Attribute
    - Innerhalb einer Methode kann mit Hilfe der EXPOSE-Anweisung, die der ::METHOD-Direktive **unmittelbar** folgen **muß**, eine Liste von Attributen definiert werden. Daran anschließend kann von **innerhalb** der entsprechenden Methode aus direkt darauf zugegriffen werden.
    - Definition und Zugriff auf Attribute kann unabhängig davon auch mit einer ATTRIBUTE-Methode erfolgen

Geltungsbereich (6)

- "**Method Scope**" (Fortsetzung)
  - Legt fest, auf *welche* Attribute Methoden *direkt* zugreifen können
  - Es gibt zwei Arten von Geltungsbereiche, die den Zugriff auf Variable (Attribute) regeln
    - Attribute, die in Klassen definiert wurden
      - gedacht für Instanzen von Klassen, daher manchmal auch als "**Instanzattribute**" bezeichnet
    - Attribute, die für "freilaufende Methoden" definiert wurden
      - Methoden, die **vor** der ersten Klassendirektive definiert wurden, besitzen einen eigenständigen Geltungsbereich für Attribute
      - Zugriff auf derartige Methoden erfolgt über die Umgebungsvariable **.METHODS**

Geltungsbereiche im Überblick

- Rexx und Object Rexx
  - Standard Scope
    - Sprungmarken, Variable
  - Procedure Scope
    - Variable in Prozeduren/Funktionen
- Object Rexx
  - Program Scope
    - Zugriff auf lokale sowie auf öffentliche Klassen und Routinen von aufgerufenen bzw. aufgesuchten (::REQUIRES) Programmen
  - Routine Scope
    - Geltungsbereich einer Routine (Standard+Procedure+Program)
  - Method Scope
    - Geltungsbereich einer Methode (Standard+Procedure+Program) und Sichtbarkeit von Attributen
      - Instanzmethoden: Methoden, die direkt für eine Klasse definiert sind ("Instanzattribute")
      - Freilaufende Methoden: Methoden ***vor*** der ersten Klassendirektive ("Freilaufattribute")

Abstrakter Datentyp "Person"
Umsetzung in Object Rexx, 1

```
/**/
p1 = .Person~New; p1~VorName= "Albert";
p1~FamilienName= "Einstein"; p1~Gehalt=45000
p2=.Person~New~~"VorName="("Vera")~~"Gehalt="("25000")
p2~~"FamilienName="("Mitirgendeinemnamen")
SAY p1~VorName p1~FamilienName p1~Gehalt
SAY p2~VorName p2~FamilienName p2~Gehalt
SAY "Lohnkosten:" p1~Gehalt + p2~Gehalt

::CLASS  Person
::METHOD Vorname       ATTRIBUTE
::METHOD Familienname  ATTRIBUTE
::METHOD Gehalt        ATTRIBUTE
```

Ausgabe:

```
Albert Einstein 45000
Vera Mitirgendeinemnamen 25000
Lohnkosten: 70000
```

Abstrakter Datentyp "Person" Umsetzung in Object Rexx, 2

```
/**/
p1 = .Person~New; p1~VorName= "Albert";
p1~FamilienName= "Einstein"; p1~Gehalt= "45000"
p2=.Person~New~~"VorName="("Vera")~~"Gehalt="(25000)
p2~~"FamilienName="("Mitirgendeinemnamen")
SAY p1~VorName p1~FamilienName p1~Gehalt p2~VorName
SAY p1~VorName p1~Gehalt p1~~erhoeheGehalt(10000)~Gehalt
::CLASS  Person
::METHOD Vorname        ATTRIBUTE
::METHOD Familienname   ATTRIBUTE
::METHOD Gehalt         ATTRIBUTE
::METHOD erhoeheGehalt
  EXPOSE Gehalt
  USE ARG Erhoehung
  Gehalt = Gehalt + Erhoehung
```

Ausgabe:

```
Albert Einstein 45000 Vera
Albert 45000 55000
```

Anlegen von neuen Objekten

- Anlegen von neuen Objekten
  - NEW-Nachricht wird der Klasse geschickt
  - Resultat ist eine ***Referenz*** *auf ein Objekt* (auf eine Instanz) der entsprechenden Klasse
- **Wenn** eine Methode mit dem Namen INIT in einer Klasse definiert wurde, dann wird diese Methode aufgerufen, ehe die Kontrolle zurückgegeben wird. Dies geschieht, indem dem Objekt von der NEW-Methode aus die Nachricht INIT geschickt wird
  - Enthielt die Nachricht NEW Argumente, werden diese ***in derselben Reihenfolge*** mit der Nachricht INIT an das Objekt weitergesandt
- INIT wird auch als ***Kon****struktor*(methode) bezeichnet

Abstrakter Datentyp "Person"
Umsetzung in Object Rexx, Konstruktor

```
/**/
p1 = .Person~New("Albert","Einstein","45000")
p2 = .Person~New("Vera","Mitirgendeinemnamen",25000)
SAY p1~VorName p1~FamilienName p1~Gehalt p2~VorName
SAY p1~VorName p1~Gehalt p1~~erhoeheGehalt(10000)~Gehalt
::CLASS  Person
::METHOD INIT
  EXPOSE  Vorname Familienname Gehalt
  USE ARG Vorname, Familienname, Gehalt
::METHOD Vorname       ATTRIBUTE
::METHOD Familienname  ATTRIBUTE
::METHOD Gehalt        ATTRIBUTE
::METHOD erhoeheGehalt
  EXPOSE Gehalt
  USE ARG Erhoehung
  Gehalt = Gehalt + Erhoehung
```

Ausgabe:

```
Albert Einstein 45000 Vera
Albert 45000 55000
```

Löschen von Objekten

- Objekte werden automatisch von Object Rexx gelöscht, sobald sie nicht mehr referenziert werden
  - DROP-Anweisung
    - Die DROP-Anweisung erlaubt das ausdrückliche Löschen einer Referenz auf ein Objekt
    - Möglichkeit besteht trotzdem, daß eine Referenz auf das Objekt in einem anderen Programmteil noch existiert
  - **Wenn** eine Methode mit dem Namen UNINIT in einer Klasse definiert wurde, dann wird diese Methode aufgerufen, unmittelbar bevor das Objekt gelöscht wird. Dies erfolgt durch den Interpreter, der dem zu löschenden Objekt die UNINIT-Nachricht schickt.
- UNINIT wird auch als ***De****struktor*(methode) bezeichnet

Abstrakter Datentyp "Person"
Umsetzung in Object Rexx, Destruktor

```
/**/
p1 = .Person~New("Albert","Einstein","45000")
p2 = .Person~New("Vera","Mitirgendeinemnamen",25000)
SAY p1~VorName p1~FamilienName p1~Gehalt p2~VorName
SAY p1~VorName p1~Gehalt p1~~erhoeheGehalt(10000)~Gehalt
DROP p1; DROP p2; CALL SysSleep( 15 ); SAY "Finish."
::CLASS  Person
::METHOD INIT
  EXPOSE  Vorname Familienname Gehalt
  USE ARG Vorname, Familienname, Gehalt
::METHOD UNINIT
  EXPOSE  Vorname Familienname Gehalt
  SAY  "Objekt: <"Vorname Familienname Gehalt"> wird gerade zerstört."
::METHOD Vorname       ATTRIBUTE
::METHOD Familienname  ATTRIBUTE
::METHOD Gehalt        ATTRIBUTE
::METHOD erhoeheGehalt
  EXPOSE Gehalt
  USE ARG Erhoehung
  Gehalt = Gehalt + Erhoehung
```

Ausgabe, zum Beispiel:

```
Albert Einstein 45000 Vera
Albert 45000 55000
Objekt: <Vera Mitirgendeinemnamen 25000> wird gerade zerstört.
Finish.
Objekt: <Albert Einstein 55000> wird gerade zerstört.
```

Abstrakter Datentyp (ADT)
Umsetzung in Object Rexx

- Abschließende Wiederholung
  - **Schema** zur Umsetzung von Datentypen
    - ::CLASS-Direktive
      - Definition von **Attributen** und damit der internen Datenstruktur
        - **EXPOSE**-Anweisung **innerhalb** von Methoden oder
        - **::**METHOD-Direktive mit der Kennzeichnung als
        - **ATTRIBUTE**
      - Definition von **Operationen** (Funktionen, Prozeduren)
        - **::METHOD**-Direktive
  - Instanzen von Klassen ("Objekte")
    - Einzelne, voneinander eindeutig unterscheidbare Ausprägungen desselben Typs
    - Weisen alle dieselben, in der Klasse vordefinierten Attribute und Operationen auf

Klassifikationsbaum (Generalisierungshierarchie)

- Generalisierungshierarchie, "Klassifikationsbaum"
  - Dient zur **Einordnung von Instanzen** (Objekten), z.B. in der Biologie
  - **Über- und Unterordnung von Klassen** (Schemata)
    - Untergeordnete Klassen "**erben**" die Eigenschaften der übergeordneten Klassen hinauf bis zur Wurzel
    - Untergeordnete Klassen **spezialisieren** in irgendeiner Art und Weise die übergeordneten Klassen
      - "Definieren von Unterschieden"
  - Manchmal kann es sinnvoll sein, daß eine untergeordnete Klasse direkt mehr als eine übergeordnete Klasse spezialisiert ("**Mehrfachvererbung**")
    - Beispiel: Klassen für die Repräsentation von Land- und Wassertieren, wobei eine Klasse für Amphibientiere direkt die Eigenschaften von Land- und Wassertieren erben könnte

Object Rexx: Klassifikationsbaum, 1

- Vorgefertigter "Klassifikationsbaum"
  - Wurzelklasse von Object Rexx hat die Bezeichnung "Object"
  - Klassendefinitionen werden als Spezialisierungen der Klasse "Object" angesehen, wenn nichts anderes angegeben ist
  - Einfach- und Mehrfachvererbung möglich
- Suchreihenfolge
  - Wenn eine Methode in der Klasse selbst nicht gefunden wird, dann wird die Suche danach aufgrund einer Nachricht in der direkt übergeordneten Klasse fortgeführt
  - Wenn eine Methode auch in der Wurzelklasse "Object" nicht gefunden wird, dann wird eine entsprechende Fehlerausnahme erzeugt
    - Wurde eine Methode UNKNOWN definiert, so wird in einem solchen Fall statt eine Fehlerausnahme diese Methode aktiviert, die unter anderem auch allfällige Argumente der ursprünglichen Nachricht erhält

Object Rexx: Klassifikationsbaum, 2

- Suchreihenfolge (Fortsetzung)
  - Für die Suchreihenfolge sind spezielle, vorbelegte Variablen **nur innerhalb von Methoden** zugänglich
    - super
      - Enthält immer eine Referenz auf die übergeordnete Klasse
      - Damit kann man z.B. eine Nachricht an übergeordnete Klassen absenden, sodaß übergeordnete Methoden aufgerufen werden
    - self
      - Enthält immer eine Referenz auf das Objekt selbst
      - Damit kann man z.B. eine weitere Nachricht an das Objekt selbst senden und somit eine weitere Methode aufrufen
  - Sowohl super als auch self bestimmen die Klasse, in der mit der Suche nach der Methode begonnen wird, die dieselbe Bezeichnung wie die Nachricht trägt

Beispiel "Hund", 1

- Problemstellung
  - Tierschutzverein haltet Hunde
    - Normale Hunde
    - Kleine Hunde
    - Große Hunde
  - Alle Hunde besitzen einen Namen und können bellen
    - Normale Hunde bellen "Wuff Wuff"
    - Kleine Hunde bellen "wuuf"
    - Große Hunde bellen "WUFFF! WUFFF!! WUFFF!!!"
  - Es sollen entsprechende Klassen definiert werden, wobei die Vererbung (die Suchreihenfolge) ausgenutzt werden soll

Beispiel "Hund", 2

- Definition einer Klasse "**Hund**", die alle Eigenschaften aufweist, die allen Hunden gemeinsam ist

```
/**/
h1 = .Hund ~NEW ~~"NAME="("Sweety") ~~bellen
```

```
::CLASS  Hund
::METHOD Name          ATTRIBUTE
::METHOD Bellen
  SAY self~Name":" "Wuff Wuff"
```

Ausgabe:

```
Sweety: Wuff Wuff
```

Beispiel "Hund", 3

- Definition einer Klasse "**GrosserHund**", die alle Eigenschaften aufweist, die allen großen Hunden gemein ist

```
/**/
h1 = .Hund    ~NEW ~~"NAME="("Sweety")  ~~bellen
.GrosserHund ~NEW ~~"NAME="("Grobian") ~~bellen
::CLASS  Hund SUBCLASS Object
::METHOD Name          ATTRIBUTE
::METHOD Bellen
  SAY self~Name":" "Wuff Wuff"
::CLASS  GrosserHund SUBCLASS Hund
::METHOD Bellen
  SAY self~Name":" "WUFFF! WUFFF!! WUFFF!!!"
```

Ausgabe:

```
Sweety: Wuff Wuff
Grobian: WUFFF! WUFFF!! WUFFF!!!
```

Beispiel "Hund", 4

- Definition einer Klasse "**KleinerHund**", die alle Eigenschaften aufweist, die allen kleinen Hunden gemein ist

```
/**/
.Hund~NEW        ~~"NAME="("Sweety")  ~~bellen
.GrosserHund~NEW ~~"NAME="("Grobian") ~~bellen
.KleinerHund~NEW ~~"NAME="("Arnie")   ~~bellen
::CLASS  Hund
::METHOD Name          ATTRIBUTE
::METHOD Bellen
  SAY self~Name":" "Wuff Wuff"
::CLASS  GrosserHund SUBCLASS Hund
::METHOD Bellen
  SAY self~Name":" "WUFFF! WUFFF!! WUFFF!!!"
::CLASS  KleinerHund SUBCLASS Hund
::METHOD Bellen
  SAY self~Name":" "wuuf"
```

Ausgabe:

```
Sweety: Wuff Wuff
Grobian: WUFFF! WUFFF!! WUFFF!!!
Arnie: wuuf
```

Beispiel "Hund", 5

- Definition einer Klasse "**KleinerHund**", die alle Eigenschaften aufweist, die allen kleinen Hunden gemein ist

```
/**/
.Hund~NEW        ~~"NAME="("Sweety")  ~~bellen
.GrosserHund~NEW ~~"NAME="("Grobian") ~~bellen
.KleinerHund~NEW ~~"NAME="("Arnie")   ~~bellen
::CLASS  Hund           SUBCLASS Object
::METHOD Name           ATTRIBUTE
::METHOD Bellen
  SAY self~Name":" "Wuff Wuff" "-" self
::CLASS  GrosserHund    SUBCLASS Hund
::METHOD Bellen
  SAY self~Name":" "WUFFF! WUFFF!! WUFFF!!!" "-" self
::CLASS  "KleinerHund" SUBCLASS Hund
::METHOD Bellen
  SAY self~Name":" "wuuf" "-" self
```

Ausgabe:

```
Sweety: Wuff Wuff - a HUND
Grobian: WUFFF! WUFFF!! WUFFF!!! - a GROSSERHUND
Arnie: wuuf - a KleinerHund
```

Nebenläufigkeiten

- Nebenläufigkeit
  - Mehrere Teile eines Programms/Programmsystems laufen zur *selben Zeit* (parallel) ab
  - Mögliche Probleme
    - Datenintegrität (Objektintegrität)
    - Deadlock-Sperren
- Object Rexx
  - **Inter**-Objekt-Nebenläufigkeiten
    - *Verschiedene* Objekte ein- und derselben Klasse sind voreinander geschützt und können zur selben Zeit aktiv sein
  - **Intra**-Objekt-Nebenläufigkeiten
    - ***Innerhalb*** einer Instanz (eines Objekts) könnten verschiedene Methoden gleichzeitig aktiv sein, *standardmäßig* aber nur dann, wenn sie in *verschiedenen Klassen* definiert sind