Abteilung für Wirtschaftsinformatik

Automatisierung von Windows Anwendungen (6)

Object Rexx Umgebungen (.local, .environment),
Klassenmethoden, Klassenattribute, Object Rexx-Klasse "Class",
Definition von Klassen und Methoden zur Laufzeit,
"One-off Objekte", Das "Große Bild",
Nebenläufigkeit, Security Manager, Unknown, Forward

Prof. Dr. Rony G. Flatscher

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

Object Rexx Umgebung (Environment), 1

- Object Rexx-Programme
  - Abarbeitungsreihenfolge
    - Syntaxüberprüfung
    - Abarbeitung von Direktiven
    - Start des Object Rexx-Programms
  - Mögliche Fragen
    - Wie werden aufgefundene Routinen, Methoden und Klassen zur Verfügung gestellt?
    - Wie werden öffentliche Routinen und Klassen sichtbar gemacht?
    - Gibt es eine Möglichkeit für Rexx-Programm(teile), Objekte mteinander zu teilen (Kopplung)?

Object Rexx Umgebung (Environment), 2

- Der Interpreter baut vier Verzeichnisobjekte (vom Typ: Directory) auf
  - Das Verzeichnis "Source"
    - Enthält alle einem Programm/Modul zur Verfügung stehenden Routinen, Methoden und Klassen
      - Im Programm/Modul definierte Routinen, Methoden und Klassen
      - Öffentliche Routinen und Klassen eines aufgerufenen (CALL oder ::REQUIRES-Direktive) Programms/Moduls
      - .METHODS, ein Verzeichnis der **"freilaufenden"** Methoden oder unbelegt (Zeichenkette: ".METHODS")
      - Nicht zugänglich gemacht, nur für das Laufzeitsystem verfügbar
    - Für jedes Programm/Modul wird vom Laufzeitsystem aus ein **individuelles** "Source"-Verzeichnis aufgebaut und gepflegt!
      - Die öffentlichen Routinen und Klassen **des zuletzt aufgerufenen Programms/Moduls ersetzen alle** entsprechenden öffentlichen Routinen und Klassen von zuvor aufgerufenen Programmen/Modulen

Object Rexx Umgebung (Environment), 3

- Das lokale Verzeichnis "Local"
  - Über das Umgebungssymbol **.LOCAL** zugänglich
  - Wird für jeden Prozeß eigens angelegt
  - Enthält prozeßbezogene Objekte, z.B.

    .error (Monitorobjekt für Fehlermeldungen),
    .input (Monitorobjekt für Eingaben),
    .output (Monitorobjekt für Ausgaben)
    [Monitorobjekte erlauben den Austausch der überwachten Objekte!]

    .stderr (Streamobjekt , das .error überwacht),
    .stdin (Streamobjekt , das .input überwacht),
    .stdout (Streamobjekt , das .output überwacht):

Object Rexx Umgebung (Environment), 4

- Das Verzeichnis "Environment"
  - Über das Umgebungssymbol **.ENVIRONMENT** zugänglich
    - Wird für jeden Prozeß eigens angelegt
  - Enthält eine Reihe wichtiger Objekte, z.B.
    - ***Sämtliche*** öffentlichen Klassen von Object Rexx
    - Grundlegenden Objekte, z.B.

      .nil das "Nichtobjekt" ("Nil" ist Englisch für: "Nichts"!): zeigt an, dass kein Objekt verfügbar ist

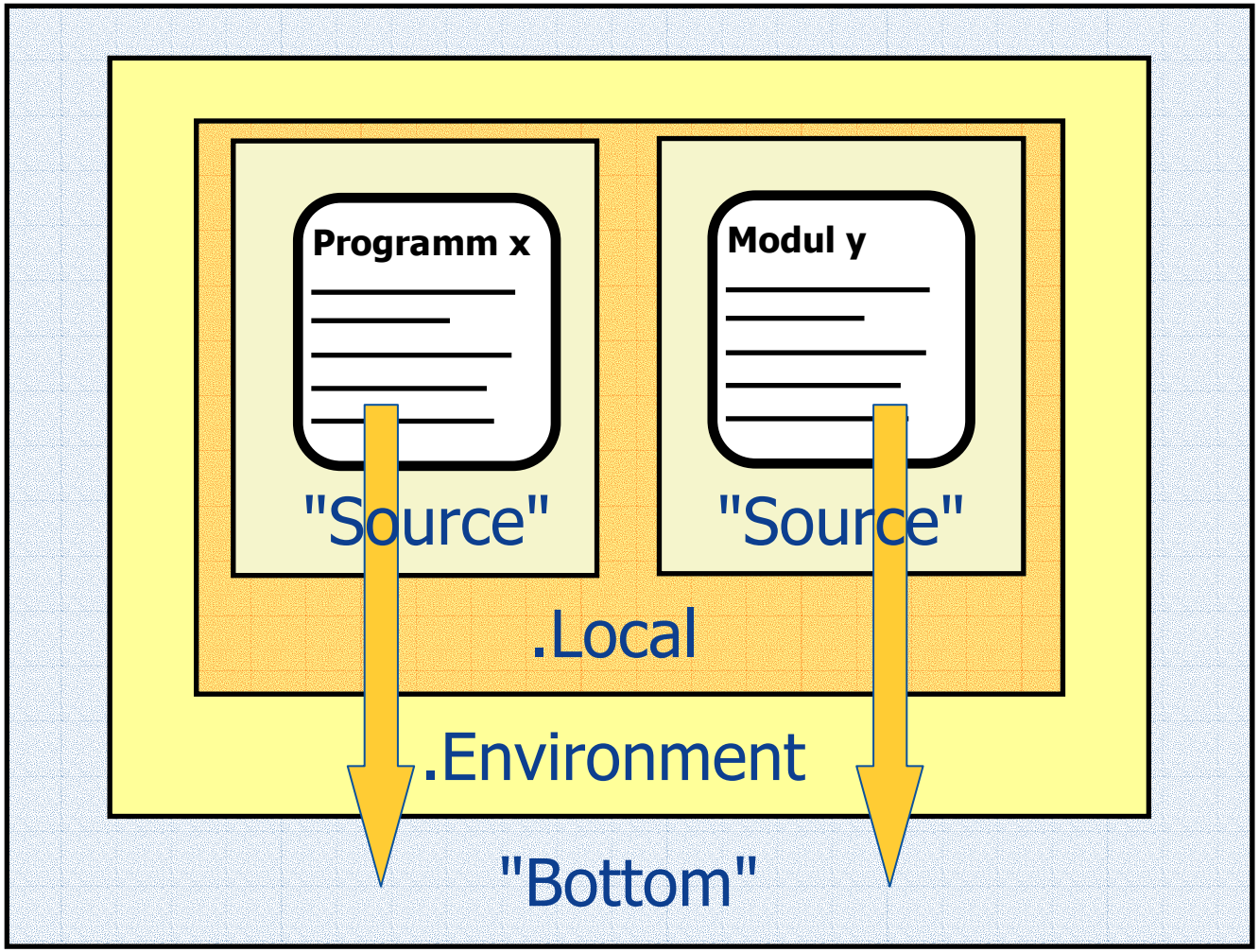
      .true der Boole'sche Wert "1" für wahr

      .false der Boole'sche Wert "0" für falsch
- Das Verzeichnis "Bottom"
  - Weitere, vom Laufzeitsystem benötigte Objekte
  - Nicht zugänglich gemacht, nur für das Laufzeitsystem verfügbar

Umgebungssymbole (Environment Symbols), 1

- Bezeichner beginnen immer mit einem Punkt
  - Systemseitige Umgebungssymbole des Interpreters weisen im Bezeichner keinen weiteren Punkt auf
  - Benutzerdefinierte Umgebungssymbole sollen per Konvention einen weiteren Punkt mitten in ihrem Bezeichner aufweisen
- Suche in der Umgebung
  - **Laufzeitsystem** entfernt den Punkt am Anfang des Bezeichners und **durchsucht alle Umgebungsverzeichnisse** in folgender fixierten Reihenfolge:
    1. Source-Verzeichnis
    2. Local-Verzeichnis (als .local zugänglich)
    3. Environment-Verzeichnis (als .environment zugänglich)
    4. Bottom-Verzeichnis

Umgebungssymbole (Environment Symbols), 2



Umgebungssymbole (Environment Symbols), 3

  - Wenn kein Eintrag mit der Bezeichnung des Umgebungssymbols zur Verfügung steht, dann wird als Ergebnis das in Großbuchstaben übersetzte Umgebungssymbol (eine Zeichenkette) zurückgegeben

- **Warnung!**
  - Keine Klassen definieren, die dieselben Bezeichnungen wie die eingebauten aufweisen ("Source"-Verzeichnis wird *vor* dem .Environment-Verzeichnis aufgesucht!)
  - **.nil**, **.true** und **.false** ***nicht*** umdefinieren!
- Weitere Informationen, z.B.

  `http://wi.wu-wien.ac.at/rgf/rexx/orx07/Local.pdf`

Object Rexx-Umgebung
Beispielsausgabe von .LOCAL

```
/* Ausgabe der Einträge des lokalen Verzeichnisses */
 tmpArr = sort(.local)
 DO entry OVER tmpArr
    SAY LEFT(entry, 25, '.') .local~at(entry)~string
 END
 ::REQUIRES sort_util.cmd /* aus "ORX8.ZIP", zum Sortieren */
```

Ausgabe:

```
ERROR..................... a Monitor
INPUT..................... a Monitor
LOCALSERVER............... a server
NLS.DEFAULT.ALIAS........ iso8859-1
OUTPUT.................... a Monitor
STDERR.................... STDERR
STDIN..................... STDIN
STDOUT.................... STDOUT
STDQUE.................... SESSION
```

Sortiermodul verfügbar unter: <http://wi.wu-wien.ac.at/rgf/rexx/orx08/>

Metaklassen, Klassenattribute, Klassenmethoden

- Metaklassen
  - Object Rexx Klasse Class und ihre Subklassen
  - Erlauben das Verwalten von Methoden, die Instanzen der Klasse besitzen sollen (Methoden DEFINE, DELETE, METHOD, METHODS)
  - Ermöglicht das Instanziieren von Objekten (Methode NEW)
  - Instanz einer Metaklasse heisst "Klassenobjekt"
  - Die Klasse, aus der ein Objekt stammt (instanziiert wurde), kann mit der Nachricht CLASS erfragt werden (liefert das entsprechende Klassenobjekt)
- Klassenattribute
  - Attribute von Metaklassen
- Klassenmethoden
  - Methoden von Metaklassen

Klassenmethoden, 1

- Klassenmethoden
  - Methoden von Metaklassen
    - Sämtliche Methoden der Object Rexx Klasse Class, z.B.
      - ID, DEFINE, DELETE, METHODS
    - Sämtliche Methoden der Object Rexx Klasse Object, z.B.
      - STRING, HASMETHOD
    - Sämtliche Methoden, die mit der ::METHOD-Direktive und dem Schlüsselwort CLASS definiert sind, werden dem Klassenobjekt hinzugefügt
      - Der Interpreter benutzt dafür die Methode SETMETHOD aus der Klasse Object
  - Klassenmethoden stehen allen Objekten zur Verfügung
    - Jedes Objekt kann seine Klasse (genauer: sein Klassenobjekt) zur Laufzeit mit der Nachricht Class erhalten und ihr anschließend die entsprechenden Nachrichten schicken

Klassenmethoden, 2

```
/**/
SAY COPIES("-", 50)
.Test~Hallo_1
SAY COPIES("-", 50)
o = .Test~New
o~Hallo_2
::CLASS  Test
::METHOD Init  CLASS
  SAY "Neue Klasse [" || self~string || "] wird angelegt..."
  self~init:super
::METHOD Hallo_1 CLASS
  SAY "Hallo, ich bin [" || self~string || "]..."
::METHOD Init
  SAY "Neue Instanz ["self~string"] wird angelegt..."
  self~init:super
::METHOD Hallo_2
  SAY "Hallo, ich bin ["self~string"]..."
```

Ausgabe:

```
Neue Klasse [The TEST class] wird angelegt...
------------------------------------------------
Hallo, ich bin [The TEST class]...
------------------------------------------------
Neue Instanz [a TEST] wird angelegt...
Hallo, ich bin [a TEST]...
```

Klassenmethoden, 3

```
/**/
SAY COPIES("-", 50)
.Test~Hallo_1
SAY COPIES("-", 50)
o = .Test~New
o~Hallo_2
::CLASS  Test
::METHOD Init  CLASS
  SAY "Neue Klasse [" || self~string || "] wird angelegt..."
  self~init:super
::METHOD Hallo_1 CLASS
  SAY "Hallo, ich bin [" || self~string || "]..."
::METHOD Init
  SAY "Neue Instanz ["self~string"] wird angelegt..."
  self~init:super
::METHOD Hallo_2
  SAY "Hallo, ich bin ["self~string"]..."
```

Ausgabe:

```
Neue Klasse [The TEST class] wird angelegt...
--------------------------------------------------
Hallo, ich bin [The TEST class]...
--------------------------------------------------
Neue Instanz [a TEST] wird angelegt...
Hallo, ich bin [a TEST]...
```

Klassenmethoden, 4

- Klassenattribute
  - Attribute von Metaklassen
    - Zugriff über Attributmethoden
    - Zugriff mit Hilfe der EXPOSE-Anweisung als erste Anweisung in einer Klassenmethode
  - Attribute von Klassen können indirekt über entsprechende Methoden zum Zugreifen und Zuweisen allen Objekten zur Verfügung gestellt werden
    - Jedes Objekt kann sein Klassenobjekt zur Laufzeit mit der Nachricht Class erhalten

Klassenmethoden, 5

```
/**/
.Test ~~New ~~New ~~New ~~New ~~New ~~New
SAY "Es wurden bisher [".Test~Zaehler"] Objekte angelegt."
o = .Test~New
SAY "Es wurden bisher ["o~class~Zaehler"] Objekte angelegt."
SAY "Klasse:" o~class~string", letzte Instanz davon:" o~string
```

```
::CLASS  Test
::METHOD Init                 CLASS
  self~Zaehler = 0
  self~init:super
::METHOD Zaehler  ATTRIBUTE CLASS
::METHOD New                  CLASS
  EXPOSE Zaehler
  Zaehler = Zaehler + 1
  FORWARD CLASS (super)
```

Ausgabe:

```
Es wurden bisher [6] Objekte angelegt.
Es wurden bisher [7] Objekte angelegt.
Klasse: The TEST class, letzte Instanz davon: a TEST
```

Metaklasse, 1

- Aufgrund einer ::CLASS-Direktive legt der Interpreter eine Instanz von **Class** ("Klassenobjekt") an
- Aufgrund der ::METHOD-Direktive(n) erzeugt der Interpreter entsprechende Methodenobjekte mit Hilfe der Klasse **Method**
  - Methodenobjekte, die für Instanzen der Klasse gedacht sind, werden als "**Instanz**methoden" bezeichnet und mit Hilfe von DEFINE-Nachrichten im Klassenobjekt gespeichert
    - Die METHODS-Nachricht liefert sämtliche Instanzmethoden
  - Methodenobjekte, die für ein Klassenobjekt selbst gedacht sind, werden als "**Klassen**methoden" bezeichnet und mit Hilfe von SETMETHOD-Nachrichten direkt dem Klassenobjekt hinzugefügt

Metaklasse, 2

- Object Rexx-Programme können so abgefaßt werden, daß erst zur Laufzeit - also **dynamisch** - sowohl Klassenobjekte als auch Methoden erzeugt werden
  - Klassen werden durch Klassenobjekte (Instanzen der Object Rexx Klasse **Class**) repräsentiert
  - Methoden werden durch Methodenobjekte (Instanzen der Object Rexx Klasse **Method**) repräsentiert
    - Instanzmethoden werden mit Hilfe der DEFINE-Nachricht im Klassenobjekt gespeichert (DEFINE ist in der Object Rexx Klasse **Class** definiert, vgl. auch DELETE)
      - METHODS liefert sämtliche Instanzmethoden, die für Instanzen definiert sind
    - Klassenmethoden können mit Hilfe der SETMETHOD-Nachricht existierenden Klassenobjekten *direkt* hinzugefügt werden (SETMETHOD ist in der Object Rexx Klasse **Object** definiert, vgl. auch UNSETMETHOD)

Metaklasse, 3

- Die Metaklasse **Class** ist eine normale Object Rexx Klasse
  - Kann daher spezialisiert werden
  - *Alle Subklassen von **Class** sind selbst auch Metaklassen !*
  - Sollen spezialisierten Metaklassen für die Erzeugung der Klassenobjekte herangezogen werden, muß in der ::CLASS-Direktive das Schlüsselwort METACLASS gefolgt von der Bezeichnung der entsprechenden Metaklasse angegeben werden
    - Die Vorbelegung lautet: METACLASS Class
- Mit Hilfe der Class-Nachricht (von **Object**) wird immer das zugehörige Klassenobjekt (Instanz einer Metaklasse) geliefert
  - Die Methoden von Metaklassen stehen daher den Objekten *immer* über das zugehörige Klassenobjekt zur Verfügung

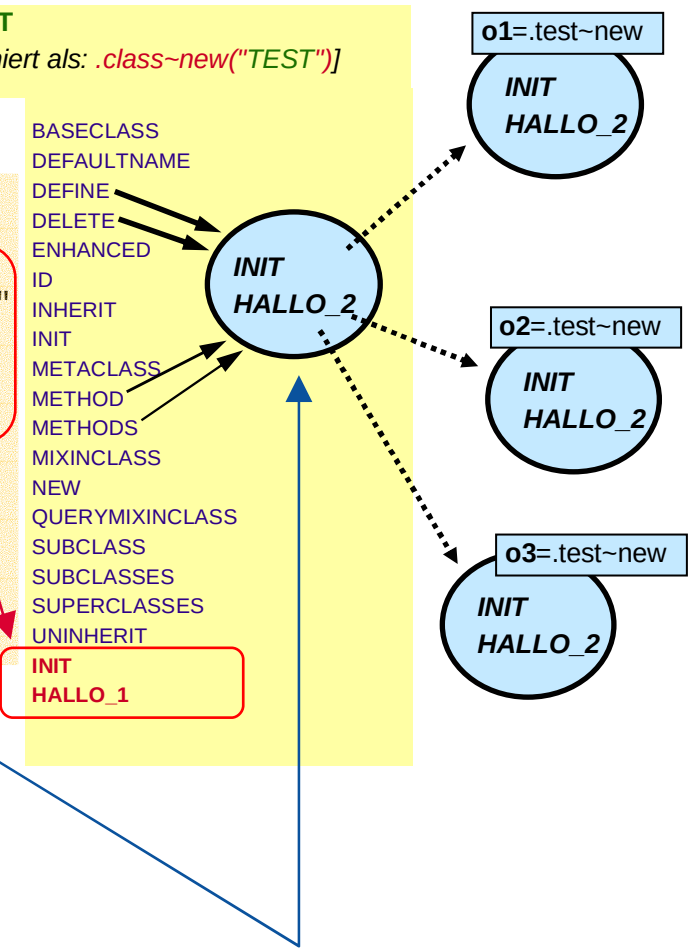
Klassen- vs. Instanzmethoden, 1
Beispiel mit Klassenobjekt ".TEST"

```
/**/
SAY COPIES("-", 50)
.Test~Hallo_1
SAY COPIES("-", 50)
o = .Test~New
o~Hallo_2

::CLASS  Test

::METHOD Init  CLASS
  SAY "Neue Klasse [" || self~string || "] wird angelegt..."
  self~init:super
::METHOD Hallo_1 CLASS
  SAY "Hallo, ich bin [" || self~string || "]..."

::METHOD Init
  SAY "Neue Instanz ["self~string"] wird angelegt..."
  self~init:super
::METHOD Hallo_2
  SAY "Hallo, ich bin ["self~string"]..."
```



Ausgabe:

```
Neue Klasse [The TEST class] wird angelegt...
------------------------------------------------
Hallo, ich bin [The TEST class]...
------------------------------------------------
Neue Instanz [a TEST] wird angelegt...
Hallo, ich bin [a TEST]...
```

Klassen- vs. Instanzmethoden, 2
Beispiel mit Klassenobjekt ".TEST"

Direktiven des Programmes werden vom Interpreter befolgt

Umsetzen der Direktiven durch den Interpreter

```
::CLASS  Test                                  -- Klasse

::METHOD Init  CLASS                           -- Methode #1
  SAY "Neue Klasse [" || self~string || "] wird angelegt..."
  self~init:super

::METHOD Hallo_1 CLASS                         -- Methode #2
  SAY "Hallo, ich bin [" || self~string || "]..."

::METHOD Init                                  -- Methode #3
  SAY "Neue Instanz ["self~string"] wird angelegt..."
  self~init:super

::METHOD Hallo_2                               -- Methode #4
  SAY "Hallo, ich bin ["self~string"]...“
```

```
.source~test=.class~new("TEST")  -- Klasse(nobjekt) erzeugen

meth1=.method~new("INIT", source1)     -- Methode #1 erzeugen
.test~setmethod("INIT", meth1)         -- Klassenobjekt hinzufügen

meth2=.method~new("HALLO_1", source2) -- Methode #2 erzeugen
.test~setmethod("HALLO_1", meth2)     -- Klassenobjekt hinzufügen

meth3=.method~new("INIT", source3)     -- Methode #3 erzeugen
.test~define("INIT", meth3)       -- als Instanzmethode definieren

meth4=.method~new("HALLO_2", source4) -- Methode #4 erzeugen
.test~define("HALLO_2", meth4)    -- als Instanzmethode definieren
```

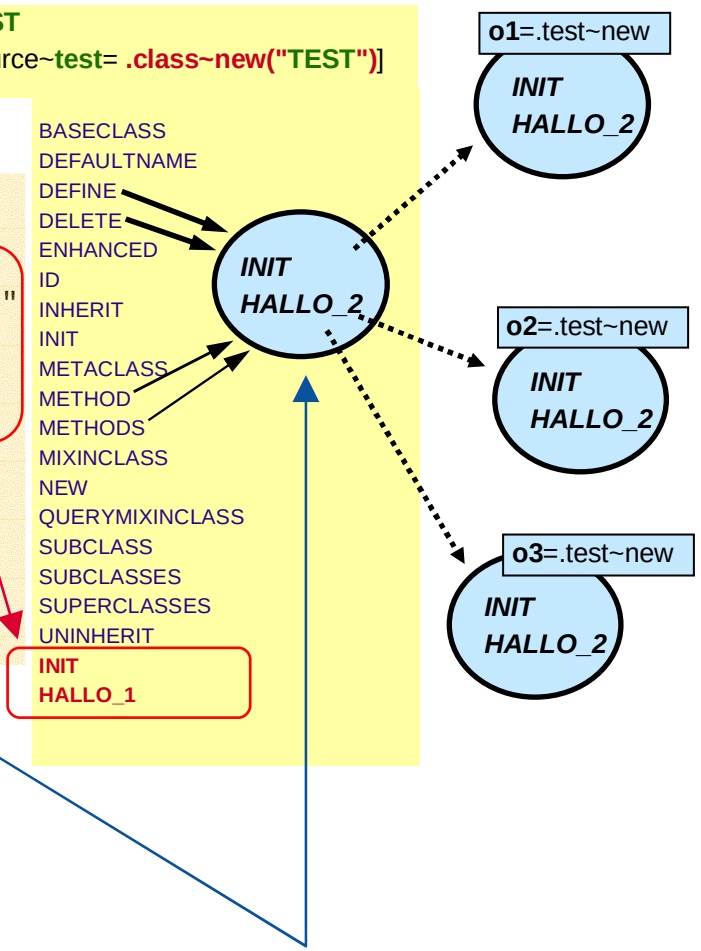
Klassen- vs. Instanzmethoden, 1
Beispiel mit Klassenobjekt ".TEST"

```
/**/
SAY COPIES("-", 50)
.Test~Hallo_1
SAY COPIES("-", 50)
o = .Test~New
o~Hallo_2

::CLASS  Test

::METHOD Init  CLASS
  SAY "Neue Klasse [" || self~string || "] wird angelegt..."
  self~init:super
::METHOD Hallo_1 CLASS
  SAY "Hallo, ich bin [" || self~string || "]..."

::METHOD Init
  SAY "Neue Instanz ["self~string"] wird angelegt..."
  self~init:super
::METHOD Hallo_2
  SAY "Hallo, ich bin ["self~string"]..."
```



Ausgabe:

```
Neue Klasse [The TEST class] wird angelegt...
------------------------------------------------
Hallo, ich bin [The TEST class]...
------------------------------------------------
Neue Instanz [a TEST] wird angelegt...
Hallo, ich bin [a TEST]...
```

Selbsterstellte Metaklassen, 1

- Manche Probleme können mit Hilfe von Metaklassen elegant gelöst werden
  - Beispiel: **Singleton**
    - Es soll gewährleistet werden, daß von manchen Klassen jeweils *nur eine einzige Instanz* erzeugt werden kann!

Selbsterstellte Metaklassen, 2

- Das Anlegen von Instanzen erfolgt mit der Klassenmethode NEW
  - Wenn daher in der entsprechenden NEW-Methode überprüft werden kann, ob eine Instanz bereits erzeugt und zurückgegeben wurde, kann
    - Verhindert werden, daß eine weitere Instanz davon erzeugt wird, indem diese Nachricht **nicht** an die übergeordnete Klassen weitergereicht wird, **und**
    - Sofern diese einzige Instanz in einem Klassenattribut gespeichert ist, diese Instanz stattdessen zurückgegeben werden
  - Es wird eine Metaklasse Singleton definiert
    - Soll es von einer Klasse in weiterer Folge nur eine einzige Instanz geben, dann muß bei der ::CLASS-Direktive dafür **lediglich** METACLASS Singleton angegeben werden!

Selbsterstellte Metaklassen, 3

- Object Rexx-Umsetzung der Metaklasse ***Singleton***

```
/**/
::CLASS Singleton SUBCLASS Class

::METHOD Init
  EXPOSE SingleInstance
  SingleInstance = .nil
  self~init:super

::METHOD New
  EXPOSE SingleInstance
  IF SingleInstance = .nil THEN
  DO
     FORWARD CLASS (super) CONTINUE
     SingleInstance = RESULT
  END
  RETURN SingleInstance
```

Selbsterstellte Metaklassen, 4

- Für den Osterhasen wird ein abstrakter Datentyp definiert
  - Attribut: **EinsatzOrt**
  - Methoden: Wert für/von Einsatzort setzen/abfragen
- Nachdem es *nur einen **einzigen** Osterhasen* geben kann, muß sichergestellt werden, daß auch nur einer erzeugt (instantiiert, angelegt) werden kann
  - Das Klassenobjekt für **Osterhase** soll daher von der Metaklasse **Singleton** angelegt werden, die selbständig diese Bedingung sicherstellen kann

Selbsterstellte Metaklassen, 5

```
/* Osterhase */
  a = .Osterhase~new("Wien")
  b = .Osterhase~new("Stumm im Zillertal")
  SAY "a==b:" (a==b) "Einsatzort von b:" b~Einsatzort
  ::CLASS  Osterhase  METACLASS Singleton
  ::METHOD Einsatzort ATTRIBUTE
  ::METHOD Init
    self~Einsatzort = ARG(1)
    SAY "Init-Methode: Einsatzort ist:" self~Einsatzort
    self~init:super
   ::CLASS Singleton SUBCLASS Class
   ::METHOD Init
    EXPOSE SingleInstance
    SingleInstance = .nil
    self~init:super
    ::METHOD New
    EXPOSE SingleInstance
    IF SingleInstance = .nil THEN DO
       FORWARD CLASS(super) CONTINUE
       SingleInstance = RESULT
    END
    RETURN SingleInstance
```

Ausgabe:

```
Init-Methode: Einsatzort ist: Wien
a==b: 1 Einsatzort von b: Wien
```

Selbsterstellte Metaklassen, 6

- In Beispiel 2 wurde für die Testklasse die Anzahl der erzeugten Objekte gezählt
- Es soll eine Metaklasse **Counter** definiert werden, die
  - Zählt, wie oft eine Instanz erzeugt wird
  - Die Summe der bisher erzeugten Instanzen liefert

```
::CLASS  Counter   SUBCLASS Class

::METHOD Init
  self~Counter = 0
  self~init:super

::METHOD Counter  ATTRIBUTE

::METHOD New
  EXPOSE Counter
  Counter = Counter + 1
  FORWARD CLASS (super)
```

Selbsterstellte Metaklassen, 7

```
/**/
.Test ~~New ~~New ~~New ~~New ~~New ~~New
SAY "Es wurden bisher [".Test~Counter"] Objekte angelegt."
o = .Test~New
SAY "Es wurden bisher ["o~class~Counter"] Objekte angelegt."
SAY "Klasse:" o~class~string", letzte Instanz davon:" o~string
```

```
::CLASS  Test      METACLASS Counter
```

```
::CLASS  Counter   SUBCLASS Class
::METHOD Init
  self~Counter = 0
  self~init:super
::METHOD Counter  ATTRIBUTE
::METHOD New
  EXPOSE Counter
  Counter = Counter + 1
  FORWARD CLASS (super)
```

Ausgabe:

```
Es wurden bisher [6] Objekte angelegt.
Es wurden bisher [7] Objekte angelegt.
Klasse: The TEST class, letzte Instanz davon: a TEST
```

"ENHANCED"-Methode von "Class", 1

- Manchmal muß eine Instanz gebildet werden, die die Eigenschaften (Attribute, Methoden) einer bestehenden Klasse aufweisen soll, sich aber durch einige wenige Methoden davon unterscheidet
  - Problem: man muß eine eigene Klasse dafür definieren
  - Wenn zahlreiche derart leicht unterschiedliche Objekte benötigt werden, ist die Definition von Klassen mühsam
- Die ENHANCED-Methode erlaubt es, eine Instanz von einer bestehenden Klasse zu bilden und diesem Objekt jene Methoden mitzugeben, die sie anders beziehungsweise zusätzlich implementiert
  - "One-off-Objekte"

"ENHANCED"-Methode von "Class", 2

- Klasse **PERSON** verfügt über das Attribut **Name** und die Methoden **your_name** und **from_where** sowie zahlreiche weitere, sprachunabhängige Methoden
  - Es sollen die Methoden **your_name** und **from_where** für Personen unterschiedlicher Nationalität in der entsprechenden nationalen Sprache implementiert werden
    - Vorbelegung ist die deutsche Sprache: "Ich heiße ...", "Ich komme aus Österreich."
    - Englische Personen: "My name is ...", "I am from ..."
    - Französische Personen: "Je m'appelle ...", "Je vien de ..."

"ENHANCED"-Methode von "Class", 3

```
/**/
fr = .directory~new
fr ~your_name = "RETURN 'Je m''appelle' self~Name'.'"
fr ~from_where= "RETURN 'Je vien de France.' "
en = .directory~new
en ~your_name = "RETURN 'My name is' self~Name'.'"
en ~from_where= "RETURN 'I am from America.'"
p1 = .Person~new("Hans")
p2 = .Person~enhanced(fr, "Jean")
p3 = .Person~enhanced(en, "John")
SAY p1~your_name p1~from_where
SAY p2~your_name p2~from_where
SAY p3~your_name p3~from_where
::CLASS  Person
::METHOD Name      ATTRIBUTE
::METHOD Init
  self~Name = ARG(1)
::METHOD your_name
  RETURN "Ich heiße" self~Name"."
::METHOD from_where
  RETURN "Ich komme aus Österreich."
```

Ausgabe:

```
  Ich heiße Hans. Ich komme aus Österreich.
  Je m'appelle Jean. Je vien de France.
  My name is John. I am from America.
```

Das "Große Bild"
Initialisierung von Object Rexx

- Interpreter wird gestartet
- Die Object Rexx Umgebungen **.environment** und **.local** werden eingerichtet
- Rexx-Programm, das als Argument angegeben wurde
  - Wird auf Syntaxfehler hin überprüft
  - Direktiven werden befolgt, "Quell-Umgebung" **source** eingerichtet
    - ::REQUIRES lädt das angegebene Programm, führt Syntaxüberprüfung durch und befolgt die Direktiven (entsprechende "Quell-Umgebung" wird eingerichtet) die Ausführung der Programmanweisungen beginnt ab Zeile eins
    - Klassenobjekte für die Object Rexx-Klassen werden gebildet und im Quellverzeichnis gespeichert
  - Ausführung des Programmes beginnt mit den Anweisungen ab Zeile eins

Nebenläufigkeiten (Multithreading), 1

- Nebenläufigkeit
  - Parallele (gleichzeitige) Abarbeitung von verschiedenen Object Rexx-Programmen
    - Parallele Abarbeitung (Ausführung) von Methoden
      - Nebenläufigkeit **zwischen** verschiedenen Objekten: **"Inter**-Nebenläufigkeit**"**
      - Nebenläufigkeit **innerhalb** ein- und desselben Objektes: **"Intra**-Nebenläufigkeit**"**
  - Mögliche Probleme
    - Nutzung von gemeinsamen Ressourcen, z.B.
      - Gleichzeitige Veränderung von Attributen,
      - Gleichzeitige Veränderung von Dateien etc.
    - Aussperren und "Deadlocks"
      - Objekt 1 reserviert: Ressource **A** und dann **B**
      - Objekt 2 reserviert: Ressource **B** und dann **A**

Nebenläufigkeiten (Multithreading), 2

- Object Rexx-Defaultverhalten
  - Standardmäßig werden sämtliche Methoden einer Klasse durch **GUARD** serialisiert
    - Es darf *innerhalb* einer Klasse standardmäßig nur eine einzige Methode für ein- und dasselbe Objekt ausgeführt werden, da diese **exklusiv** den Zugriff auf die Attribute des Objekts **sperrt**
    - Methoden ein- und desselben Objekts, die in unterschiedlichen Klassen(ebenen) definiert sind, können parallel abgearbeitet werden (Intra-Nebenläufigkeit), da die Attribute von Objekten nach Klassen(ebenen) getrennt verwaltet werden
  - **UNGUARD** in der Methodendirektive definiert für ein- und dasselbe Objekt eine Methode als parallel ausführbar zu anderen derselben Klasse
    - ***Kein*** exklusiver *Zugriffsschutz auf die Attribute des Objektes mehr!*
    - Z.B. sinnvoll, wenn Attribute nicht verändert werden oder überhaupt kein Zugriff auf die Attribute des Objektes erfolgt

Nebenläufigkeiten (Multithreading), 3

- Object Rexx-Defaultverhalten
  - Für Methoden kann auch zur Laufzeit entschieden werden, ob sie parallel zu anderen Methoden derselben Klasse abarbeitbar sind
    - REPLY-Anweisung in einer Methode
      - Gleicher Effekt wie die RETURN-Anweisung
        - Aufrufender Programmteil erhält die Abarbeitungskontrolle zurück, **aber**
        - **zusätzlich** wird parallel dazu die Methode selbst weiter abgearbeitet!
      - Optional kann die REPLY-Anweisung über einen Rückgabewert für den aufrufenden Programmteil verfügen
        - Wenn mit der REPLY-Anweisung ein Rückgabewert angegeben wird, darf in weiterer Folge in der Methode daran anschließend keine RETURN-Anweisung Rückgabewerte besitzen

Nebenläufigkeiten (Multithreading), 4

- Für Methoden kann auch zur Laufzeit entschieden werden, ob sie parallel zu anderen Methoden derselben Klasse abarbeitbar sind
  - GUARD
    - GUARD ON-Anweisung
      - Es soll ein **exklusiver Zugriff auf die Attribute** erfolgen; wenn eine andere Methode bereits den exklusiven Zugriff darauf besitzt, wird bis zur dessen Freigabe **gewartet**
    - Die GUARD OFF-Anweisung gibt den exklusiven Zugriff auf die Attribute des Objekts wieder frei
  - Effiziente Absicherung von "kritischen Abschnitten" (englisch: "critical segments")
    - Warten auf die Zugriffssperre kann abhängig vom Auftreten bestimmter Werte in den Attributen des Objektes gemacht werden
    - Warten auf die Freigabe der Zugriffsperre kann abhängig vom Auftreten bestimmter Werte in den Attributen des Objektes gemacht werden

Nebenläufigkeiten (Multithreading), 5 Beispiel (REPLY)

```
/* */
a=.x~new
b=.x~new
c=.x~new
fifo=.fifo~new      /* FIFO-instance */
.local~repetitions = 500
a~testwrite(fifo, "from_a")
b~testwrite(fifo, "FROM_B")
c~testread(fifo)
say "after testread"
```

```
::class X

::method testwrite
  use arg fifo, msg1
  REPLY
  do i=1 to .repetitions
    fifo~write(msg1 i)
  end

::method testread
  use arg fifo
  REPLY
  do while fifo~items > 0
    i=fifo~read
    say i
  end
```

```
::class FIFO

::method init
  expose buffer
  buffer=.queue~new

::method write
  expose buffer
  use arg tmp
  buffer~queue(tmp)

::method read
  expose buffer
  return buffer~pull

::method items
  expose buffer
  return buffer~items
```

Ausgabe z.B.:

```
after testread
from_a 1
from_a 2
from_a 3
FROM_B 1
from_a 4
FROM_B 2
...
```

Nebenläufigkeiten (Multithreading), 6
Beispiel (REPLY, GUARD ON|OFF)

```
/* */
a=.x~new
b=.x~new
c=.x~new
fifo=.fifo~new      /* FIFO-instance */
.local~repetitions = 500
a~testwrite(fifo, "from_a")
b~testwrite(fifo, "FROM_B")
c~testread(fifo)
say "after testread"
```

```
::class X

::method testwrite
  use arg fifo, msg1
  REPLY
  do i=1 to .repetitions
    fifo~write(msg1 i)
  end

::method testread
  use arg fifo
  REPLY
  do while fifo~items > 0
    i=fifo~read
    say i
  end
```

```
::class FIFO
::method init
  expose buffer lock
  buffer=.queue~new
  lock=.false

::method write UNGUARDED
  expose buffer lock
  GUARD ON WHEN lock=.false
  lock=.true
  GUARD OFF
  use arg tmp
  buffer~queue(tmp)      /* queue item */
  GUARD ON
  lock=.false

::method read  UNGUARDED
  expose buffer lock
  GUARD ON WHEN lock=.false
  lock=.true; GUARD OFF
  data=buffer~pull       /* get item   */
  GUARD ON; lock=.false
  return data

::method items
  expose buffer
  return buffer~items
```

```
Ausgabe z.B.:

after testread
from_a 1
from_a 2
from_a 3
from_a 4
FROM_B 1
...
```

Nebenläufigkeiten (Multithreading), 7
Klasse MESSAGE

- Nachrichtenklasse
  - Zwei Möglichkeiten zum Versenden von Nachrichten
    - SEND - synchrone Abarbeitung
      - Abarbeitung erfolgt erst, nachdem die Nachricht abgearbeitet wurde
    - START - **a**synchrone Abarbeitung (Multithreading)
      - Nachricht wird verschickt und bewirkt das Aktivieren einer Methode
      - Die Abarbeitung des aufrufenden Programmteils erfolgt parallel dazu
  - Weitere Methoden der Nachrichtenklasse
    - COMPLETED - Gibt an, ob Nachricht abgearbeitet wurde
    - RESULT - Wartet auf und liefert das Resultat der aufgerufenen Methode zurück
    - NOTIFY - Ermöglicht es, nach dem Abarbeiten einer Nachricht eine neue Nachricht abzusenden (serialisiertes Senden von Nachrichten)

Nebenläufigkeiten (Multithreading), 8
Beispiel (MESSAGE, kein REPLY!)

```
/* */
a=.x~new
b=.x~new
c=.x~new
fifo=.fifo~new              /* FIFO-instance */
.local~repetitions = 500
.message~new(a, "testwrite", "I", fifo, "from_a")~start
.message~new(b, "testwrite", "I", fifo, "FROM_B")~start
.message~new(c, "testread",  "I", fifo)          ~start
say "after testread"


::class X

::method testwrite
  use arg fifo, msg1

  do i=1 to .repetitions
    fifo~write(msg1 i)
  end

::method testread
  use arg fifo

  do while fifo~items > 0
    i=fifo~read
    say i
  end
```

```
::class FIFO

::method init
  expose buffer
  buffer=.queue~new

::method write
  expose buffer
  use arg tmp
  buffer~queue(tmp)

::method read
  expose buffer
  return buffer~pull

::method items
  expose buffer
  return buffer~items
```

Ausgabe z.B.:

```
after testread
from_a 1
from_a 2
from_a 3
from_a 4
FROM_B 1
FROM_B 2
...
```

Security Manager, 1

- Ermöglicht das gezielte Abfangen von
  - Zugriffen auf die **Umgebung**
    - ADDRESS-Anweisung,
    - Nachrichten an .local oder .environment
  - **Externen** Funktionsaufrufen
    - Aufruf von **externen** Programmen mit der CALL-Anweisung oder mit Hilfe der ::REQUIRES-Direktive
  - Nachrichten, die als **geschützt** gekennzeichnete Methoden aktivieren
    - Schlüsselwort PROTECTED in der Methodendirektive oder
    - SETPROTECTED-Methode in der Klasse **Method**
  - Funktionsaufrufe für beziehungsweise Nachrichten an Stream-Objekte, beispielsweise
    - CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, LINES, STREAM

Security Manager, 2

- Object Rexx-Programmen kann es damit unmöglich gemacht werden - direkt oder indirekt - unerlaubte Zugriffe auf die Umgebung erfolgreich durchzuführen!
  - Abgefangene Funktionsaufrufe/Nachrichten können
    - durch **selbstdefinierte** Funktionen/Nachrichten transparent **ersetzt** werden:
      - "Transparent": das ausgeführte Programm bemerkt nicht, daß eine andere Funktion/Nachricht aufgerufen wird!
    - dazu führen, daß **kontrolliert** eine Zugriffsverletzung angezeigt und damit das Programm **abgebrochen** wird
    - zur Ausführung freigegeben werden

Security Manager, 3

- Einsatzmöglichkeiten
  - Erstellung von Ablaufprofilen für (Object) Rexx Programme
  - Erstellung von unternehmens- und aufgabenspezifischen, sicheren Umgebungen (z.B. "sandbox") für (Object) Rexx Programme
    - Sicherer Ablauf von Object Rexx-Programmen, die aus anonmyen oder unsicheren Quellen stammen, beispielsweise
      - "Roaming Agents", die über das Internet verteilt werden
  - Aufzeichnen ("Loggen") von Funktionsaufrufen/Nachrichten, die als wichtig angesehen werden

Security Manager, 4

- Vorgehensweise
  - Vom Programm wird mit Hilfe der **Method**-Nachricht NewFile ein Methodenobjekt erzeugt
  - Dem Methodenobjekt wird mit Hilfe der **Method**-Nachricht SetSecurityManager ein Security-Manager-/Supervisor-Objekt zugeordnet
  - Nach dem Aktivieren des Methodenobjektes werden vom Laufzeitsystem aus, dem Supervisor-Objekt folgende Nachrichten geschickt
    - CALL, COMMAND, REQUIRES, LOCAL, ENVIRONMENT, STREAM, METHOD
      - Gemeinsam mit jeder Nachricht wird als Argument ein Directory-Objekt übergeben, das weitere Informationen zur Auswertung enthält
      - Rückgabewert **muß** 1 (behandelt) oder 0 (weiterabarbeiten) sein

Security Manager, 5
Client-Programm ("Agent1.cmd")

```
/*================================================================*/
/* Agent Sample, "agent1.cmd"                                     */
/*================================================================*/

   interpret 'echo Hello There'
   'dir foo.bar'
   call rxfuncadd sysloadfuncs, rexxutil, sysloadfuncs
   say result
   say syssleep(1)
   say linein('c:\config.sys')
   say .array
   .object~objectname
   ::requires agent2.cmd
```

(Beispiel aus der Online-Dokumentation zu Object Rexx, Abschnitt "Security Manager" entnommen. Vgl. daher auch Programmreferenz.)

Security Manager, 6
Supervisor-Programm: "No Way!"

- Auszuführendes Programm wird mit Hilfe der **Method**-Methode NewFile in Form eines Methodenobjekts gespeichert
- Dieses Methodenobjekt wird in einem Agentenobjekt mit Hilfe der Nachricht DISPATCH aktiviert

```
/* parameter: filename of agent program */
 parse arg program                    -- parse name of file
 method = .method~newfile(program) -- create a method from the program in given file
 say "Calling program" program "with a closed cell manager:"
 pull
 method~setSecurityManager(.noWay~new) -- define which supervisor class to use
 agent = .agent~new(method) -- give instance the program to be supervised
 agent~dispatch               -- invoke program
/*-----------------------------------------------------------------------------*/
 ::CLASS Agent
 ::METHOD init                              /* Agent initialisation            */
  use arg agentmethod
  self~setmethod('DISPATCH', agentmethod) /* method available with 'dispatch'  */
/*-----------------------------------------------------------------------------*/
 ::CLASS noWay -- a supervisor class using the security manager
 ::METHOD unknown /* everything trapped by unknown and everything is an error  */
   raise syntax 98.948 array("You didn't say the magic word!")
```

(Beispiel aus der Online-Dokumentation zu Object Rexx, Abschnitt "Security Manager" entnommen. Vgl. daher auch Programmreferenz.)

Security Manager, 7
Supervisor-Programm: "Dumper"

```
/* parameter: filename of agent program */
 parse arg program
 method = .method~newfile(program) /* Read the agent program from file */
 say "Calling program" program "with an audit manager:"
 pull
 method~setSecurityManager(.Dumper~new(.output))
 agent = .agent~new(method)
 agent~dispatch
/*----------------------------------------------------------------------------*/
 ::CLASS Agent
 ::METHOD init                              /* Agent initialisation            */
  use arg agentmethod
  self~setmethod('DISPATCH', agentmethod) /* method available with 'dispatch'  */
/*----------------------------------------------------------------------------*/
 ::CLASS dumper

  ::METHOD init
    expose stream                           /* target stream for output        */
    use arg stream                          /* hook up the output stream       */

  ::METHOD unknown                          /* generic unknown method          */
    expose stream                           /* need the global stream          */
    use arg name, args                      /* get the message and arguments   */
                                            /* write out the audit event       */
    stream~lineout(time() date() 'Called for event' name)
    stream~lineout('Argments are:')         /* write out the arguments         */
    info = args[1]                          /* info directory is the first arg */
    do name over info                       /* dump the info directory         */
      stream~lineout('Item' name':' info[name])
    end
    return 0                                /* allow this to proceed           */
```

Security Manager, 8
Supervisor-Programm: "Replacer"

```
::CLASS replacer SUBCLASS noWay          /* inherit restrictive UNKNOWN method*/

::METHOD command                         /* issuing commands                  */
  use arg info                           /* access the directory              */

  info~rc = 1234                         /* set the command return code       */
  info~failure = .true                   /* raise a FAILURE condition         */
  return 1                               /* return "handled" return value     */
::METHOD call                            /* external function/routine call    */
  use arg info                           /* access the directory              */
                                         /* all results are the same          */
  info~result = "uh, uh, uh...you didn't say the magic word"
  return 1                               /* return "handled" return value     */
::METHOD stream                          /* I/O function stream lookup        */
  use arg info                           /* access the directory              */
                                         /* replace with a different stream   */
  info~stream = .stream~new('C:\OBJREXX\READ.ME')
  return 1                               /* return "handled" return value     */
::METHOD local                           /* .LOCAL variable lookup            */
                                         /* no value returned at all          */
  return 1                               /* return "handled" return value     */
::METHOD environment                     /* .ENVIRONMENT variable lookup      */
                                         /* no value returned at all          */
  return 1                               /* return "handled" return value     */
::METHOD method                          /* protected method invocation       */
  use arg info                           /* access the directory              */
                                         /* all results are the same          */
  info~result = "uh, uh, uh...you didn't say the magic word"
  return 1                               /* return "handled" return value     */
::METHOD requires                        /* REQUIRES directive                */
  use arg info                           /* access the directory              */
                                         /* switch to load a different file   */
  info~name = 'C:\OBJREXX\AGENT3.CMD'
  info~securitymanager = self            /* load under this authority         */
  return 1                               /* return "handled" return value     */
```

UNKNOWN

- Für den Fall, daß eine Methode nicht gefunden wird
  - UNKNOWN-Methode wird aufgerufen, sofern vorhanden
    - Argumente
      - Name der nicht gefundenen Methode
      - Array-Objekt mit den bereitgestellten Argumenten

```
/* Beispiel für die Definition einer UNKNOWN-Methode */
::METHOD UNKNOWN
    USE ARG meth_name, meth_args
    SAY "unknown method: ["meth_name"]“
    DO i=1 TO meth_args~items
       SAY "  arg #" i": ["i"] value: ["meth_args[i]"]“
    END
```

  - Ansonsten wird die NOMETHOD-Ausnahme aufgeworfen
    - Wenn keine Ausnahmebehandlung mit der SIGNAL ON Anweisung definiert ist, wird vom Laufzeitsystem ein Syntaxfehler erzeugt, der zum Programmabbruch führt

FORWARD-Anweisung

- "Nachrichten-Umleitung"
  - Änderung des Zielobjekts (TO)
  - Weitergabe der Nachricht an Superklassen (CLASS)
  - Änderung der Nachrichtenbezeichnung (MESSAGE)
  - Leitet erhaltene Argumente weiter, außer
    - ARGUMENT oder
    - ARRAY ist angegeben
  - Kehrt nach der Nachrichtenweiterleitung an die Stelle zurück, von der aus ursprünglich die Nachricht abgesendet wurde, außer
    - CONTINUE ist angegeben