Abteilung für Wirtschaftsinformatik

Automatisierung von Windows Anwendungen (5)

Definition von Klassen ("CLASS"-Direktive),
Definition von Methoden ("METHOD"-Direktive),
Object Rexx-Klassen, Object Rexx-Collection Klassen

Prof. Dr. Rony G. Flatscher

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

© Prof. Dr. Rony G. Flatscher

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

© Prof. Dr. Rony G. Flatscher

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

© Prof. Dr. Rony G. Flatscher

Abstrakter Datentyp (ADT)
Object Rexx (Beispiel)

- 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
```


© Prof. Dr. Rony G. Flatscher

Abarbeitung von kaskadierenden Nachrichten, 1

- Abarbeitung der kaskadierenden Nachrichten im RVALUE-Teil

```
g2=.Geburtstag~New ~~"Datum="('20080229') ~~"Time="('19:19')
```

wird vom Interpreter wie folgt ausgeführt:

```
_rvalue = .Geburtstag~New                  -- normale Nachricht
          _rvalue ~"Datum="('20080229')    -- kaskadierende Nachricht
          _rvalue ~"Time="('19:19')        -- kaskadierende Nachricht
g2=_rvalue                                 -- Zuweisung an LVALUE-Teil
```


© Prof. Dr. Rony G. Flatscher

Abarbeitung von kaskadierenden Nachrichten, 2

- Abarbeitung der kaskadierenden Nachrichten im RVALUE-Teil

```
x='20080229'~"+"(1) ~~"*"(987) ~"/"(2) ~~"+"(6) ~"+"(1)
```

wird vom Interpreter wie folgt ausgeführt:

```
_rvalue = '20080229' ~"+"(1)    -- normale Nachricht, Wert: '120080230'
          _rvalue ~"*"(987)     -- kaskadierende Nachricht
_rvalue = _rvalue ~"/"(2)       -- normale Nachricht, Wert: '10040115'
          _rvalue ~"+"(6)       -- kaskadierende Nachricht
_rvalue = _rvalue ~"+"(1)       -- normale Nachricht, Wert: '10040116'
x=_rvalue                       -- Zuweisung an LVALUE-Teil
                                -- x hat den Wert:'10040116'
```


© Prof. Dr. Rony G. Flatscher

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")

© Prof. Dr. Rony G. Flatscher

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. 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
- **Immer INIT-Methode der Superklasse aufrufen!**

© Prof. Dr. Rony G. Flatscher

Abstrakter Datentyp "Person"
INIT-Methode

```
/**/
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           -- Konstruktor
  EXPOSE  Vorname Familienname Gehalt
  USE ARG Vorname, Familienname, Gehalt
  self~init:super       -- INIT in Superklasse aufrufen
::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
```


© Prof. Dr. Rony G. Flatscher

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.
- UNINIT wird auch als ***De****struktor*(methode) bezeichnet
- **Immer UNINIT-Methode der Superklasse aufrufen, wenn sie existiert!**

© Prof. Dr. Rony G. Flatscher

Abstrakter Datentyp "Person"
UNINIT-Methode

```
/**/
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                          -- Destruktor
  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.
```


© Prof. Dr. Rony G. Flatscher

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

© Prof. Dr. Rony G. Flatscher

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

© Prof. Dr. Rony G. Flatscher

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

© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum
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

© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum
Beispiel "Hund", 2

```
/**/
.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“

::CLASS  GrosserHund SUBCLASS Hund
::METHOD Bellen
  SAY self~Name":" "WUFFF! WUFFF!! WUFFF!!!"
  self~bellen:super

::CLASS  "KleinerHund" SUBCLASS Hund
::METHOD Bellen
  SAY self~Name":" "wuuf" "-" self
```

Ausgabe:

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


© Prof. Dr. Rony G. Flatscher

Nebenläufigkeiten

- Nebenläufigkeit
  - Mehrere Teile eines Programms/Programmsystems laufen zur *selben Zeit* (parallel) ab
  - Mögliche Probleme
    - Datenintegrität (Objektintegrität)
    - Deadlock-Sperren ("Verklemmungen"), Verhungern
- 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

© Prof. Dr. Rony G. Flatscher

::CLASS-Direktive

- Entsprechend der Direktive wird vom Interpreter eine Klasse angelegt
  - **::CLASS** xyz
    - Klasse mit der Bezeichnung XYZ wird angelegt
- Schlüsselwörter bestimmen die Eigenschaften näher
  - PUBLIC
    - Optional, Klasse wird außerhalb des Moduls sichtbar
  - SUBCLASS, MIXINCLASS
    - Optional, Voreinstellung: SUBCLASS Object
  - METACLASS metaclass
    - Optional, Voreinstellung: METACLASS Class
  - INHERIT
    - Optional, erlaubt die Angabe von Klassen, von denen zusätzlich geerbt wird: **Mehrfachvererbung**

© Prof. Dr. Rony G. Flatscher

::CLASS-Direktive
Beispiel: ADT "Fahrzeug", 1

```
 /**/
.Strassenfahrzeug  ~new("LKW")  ~Fahre
.Wasserfahrzeug    ~new("Boot") ~Schwimme
```

```
::CLASS  Fahrzeug
::METHOD Bezeichnung    ATTRIBUTE
::METHOD INIT
  self~Bezeichnung = ARG(1)
```

```
::CLASS  Strassenfahrzeug   SUBCLASS Fahrzeug
::METHOD Fahre
  SAY self~Bezeichnung": 'Ich fahre jetzt...'"
```

```
::CLASS  Wasserfahrzeug     SUBCLASS Fahrzeug
::METHOD Schwimme
  SAY self~Bezeichnung": 'Ich schwimme jetzt...'"
```

Ausgabe:

```
LKW: 'Ich fahre jetzt...'
Boot: 'Ich schwimme jetzt...'
```


© Prof. Dr. Rony G. Flatscher

::CLASS-Direktive
Beispiel: ADT "Fahrzeug", 2

```
 /* Mehrfachvererbung */
.Strassenfahrzeug  ~new("LKW")  ~Fahre
.Wasserfahrzeug    ~new("Boot") ~Schwimme
.Amphibienfahrzeug ~new("Schwimmauto") ~ Zeige_Was_Du_Kannst
```

```
::CLASS  Fahrzeug
::METHOD Bezeichnung    ATTRIBUTE
::METHOD INIT
  self~Bezeichnung = ARG(1)
```

```
::CLASS  Strassenfahrzeug   MIXINCLASS Fahrzeug
::METHOD Fahre
  SAY self~Bezeichnung": 'Ich fahre jetzt...'"
```

```
::CLASS  Wasserfahrzeug     MIXINCLASS Fahrzeug
::METHOD Schwimme
  SAY self~Bezeichnung": 'Ich schwimme jetzt...'"
```

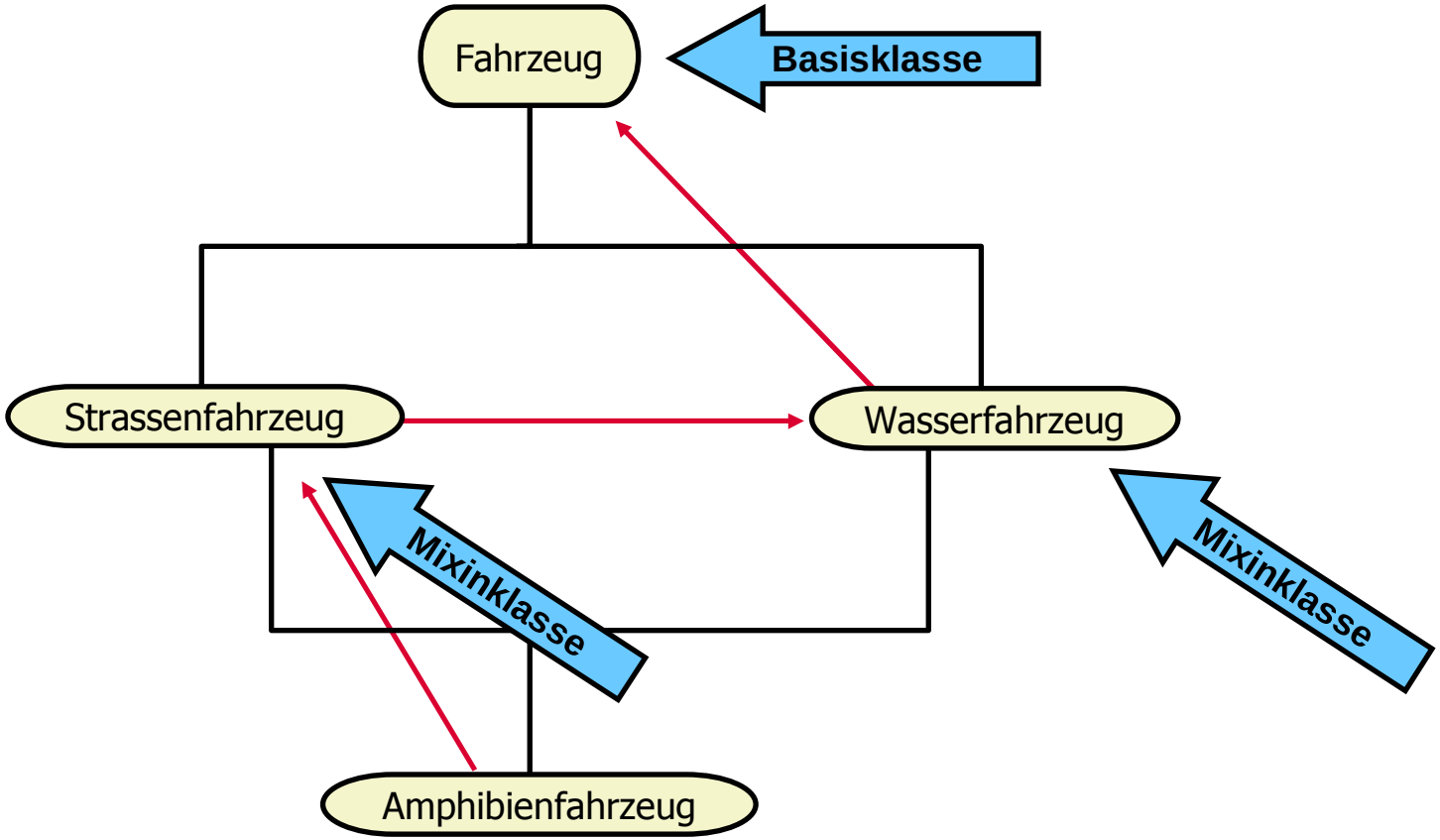
```
::CLASS Amphibienfahrzeug SUBCLASS Strassenfahrzeug INHERIT Wasserfahrzeug
::METHOD Zeige_Was_Du_Kannst
  self ~~Fahre ~~Schwimme
```

Ausgabe:

```
LKW: 'Ich fahre jetzt...'
Boot: 'Ich schwimme jetzt...'
Schwimmauto: 'Ich fahre jetzt...'
Schwimmauto: 'Ich schwimme jetzt...'
```


© Prof. Dr. Rony G. Flatscher

::CLASS-Direktive
Beispiel: ADT "Fahrzeug", 3



© Prof. Dr. Rony G. Flatscher

::METHOD-Direktive, 1

- Entsprechend der Direktive wird vom Interpreter eine Methode angelegt
  - **::Method** mmm
    - Methode mit der Bezeichnung "MMM" wird angelegt
- Schlüsselwörter bestimmen die Eigenschaften näher
  - ATTRIBUTE
    - Optional, es werden **zwei** Methoden angelegt:
      - eine Zugriffsmethode "MMM" und
      - eine Zuweisungsmethode "MMM=",
      - die beide auf das **Attribut MMM** zugreifen

© Prof. Dr. Rony G. Flatscher

::METHOD-Direktive, 2

- ATTRIBUTE (Fortsetzung)
  - *Die **Zugriff**smethode "MMM" wird immer wie folgt erzeugt:*

```
::METHOD MMM    /* Zugriffsmethode "MMM"            */
  EXPOSE MMM    /* Zugriff auf Attribut herstellen */
  RETURN MMM    /* Attributwert zurückgeben         */
```

  - *Die **Zuweisung**smethode "MMM=" wird immer wie folgt erzeugt:*

```
::METHOD "MMM=" /* Zuweisungsmethode "MMM="          */
  EXPOSE  MMM   /* Zugriff auf Attribut herstellen  */
  USE ARG MMM   /* Argumentwert dem Attribut zuweisen */
```


© Prof. Dr. Rony G. Flatscher

::METHOD-Direktive, 3

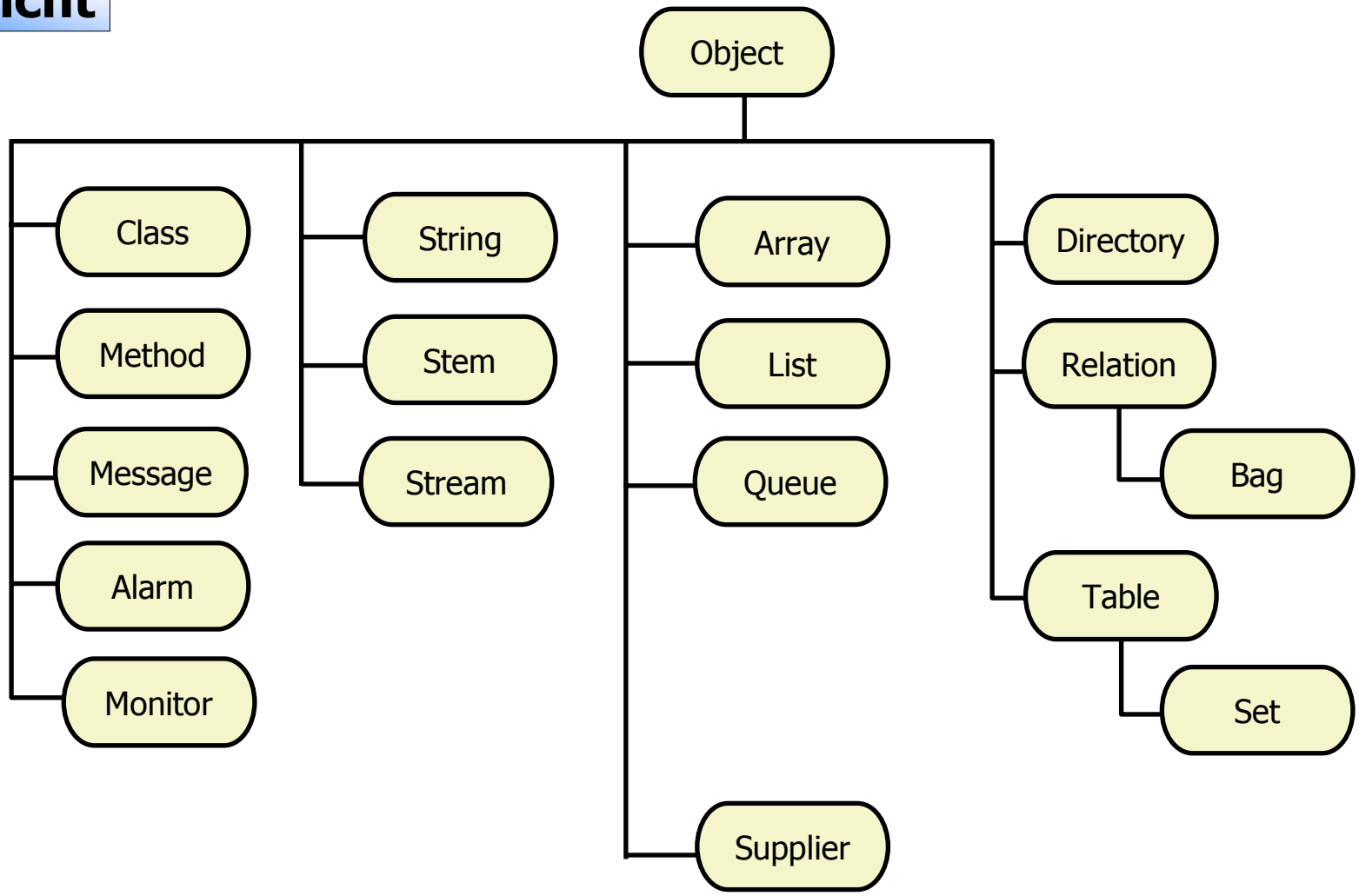
- Schlüsselwörter bestimmen die Eigenschaften näher
  - PRIVATE
    - Optional, Methode kann nur von innerhalb des Objekts aktiviert werden:

      ```
      self~mmm
      ```
  - GUARDED, UNGUARDED
    - Optional, Voreinstellung: GUARDED
    - Legt Nebenläufigkeit fest
  - CLASS
    - Optional, Methode ist eine Klassenmethode
  - PROTECTED
    - Optional, Methodenzugriff kann vom Security Manager überwacht werden

© Prof. Dr. Rony G. Flatscher

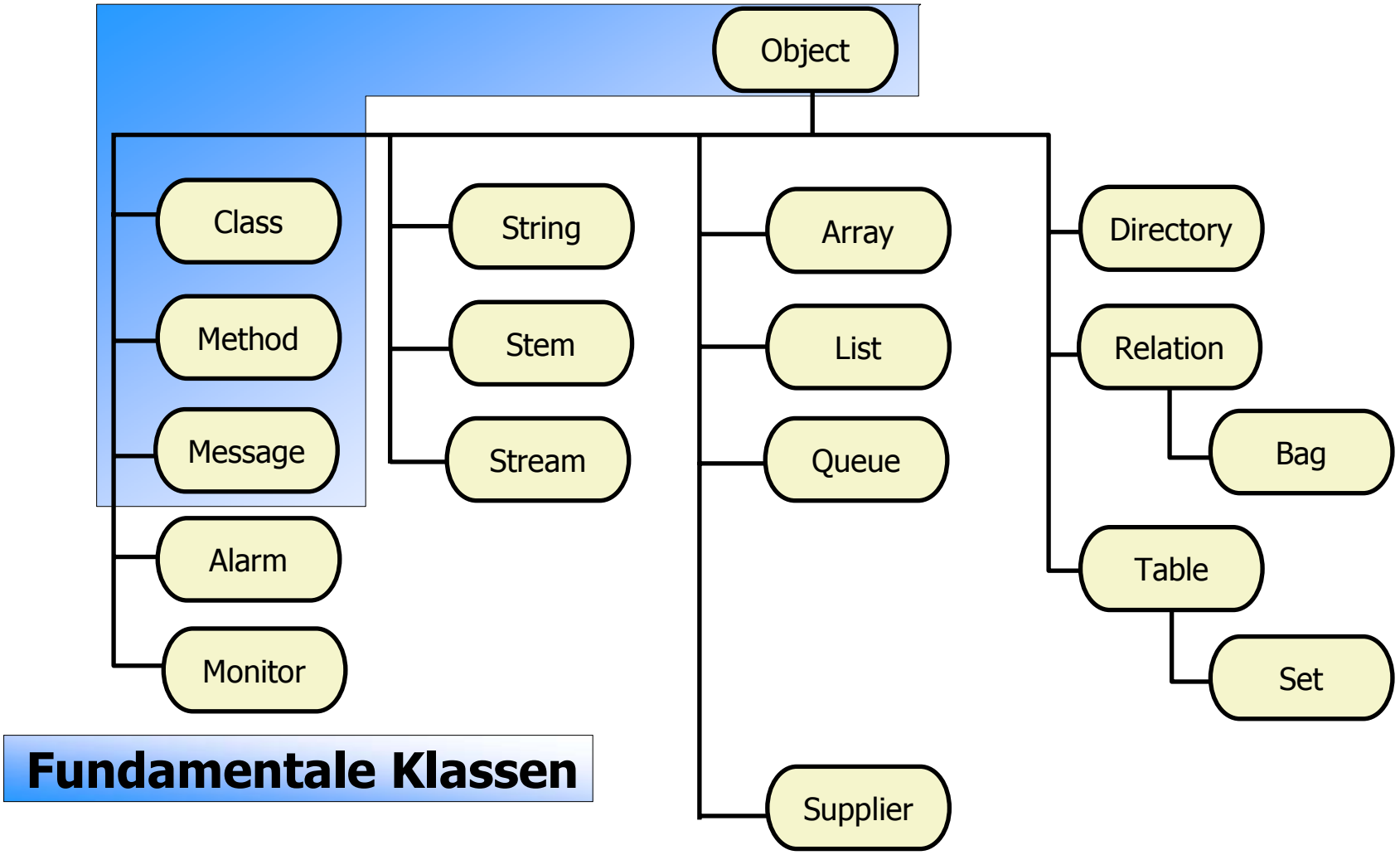
Klassifikationsbaum von Object Rexx, 1

Übersicht



© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum von Object Rexx, 2



© Prof. Dr. Rony G. Flatscher

Fundamentale Klassen, 1

- Object
  - Methoden und Attribute stehen ***allen*** Instanzen von Object Rexx-Klassen *(Objekte)* zur Verfügung
    - Beispiel: Methode INIT
      - Initialisiert ein jedes Objekt
- Class
  - Interpreter erzeugt aufgrund der ::CLASS-Direktive Instanzen *(Klassenobjekte)* von diesem Typ
    - Beispiel: Methode ID
      - Liefert den Namen (die "Identifikation") der Klasse
    - Beispiel: Methode NEW
      - Erlaubt das Erzeugen einer Instanz (eines Objektes) vom Typ der Klasse

© Prof. Dr. Rony G. Flatscher

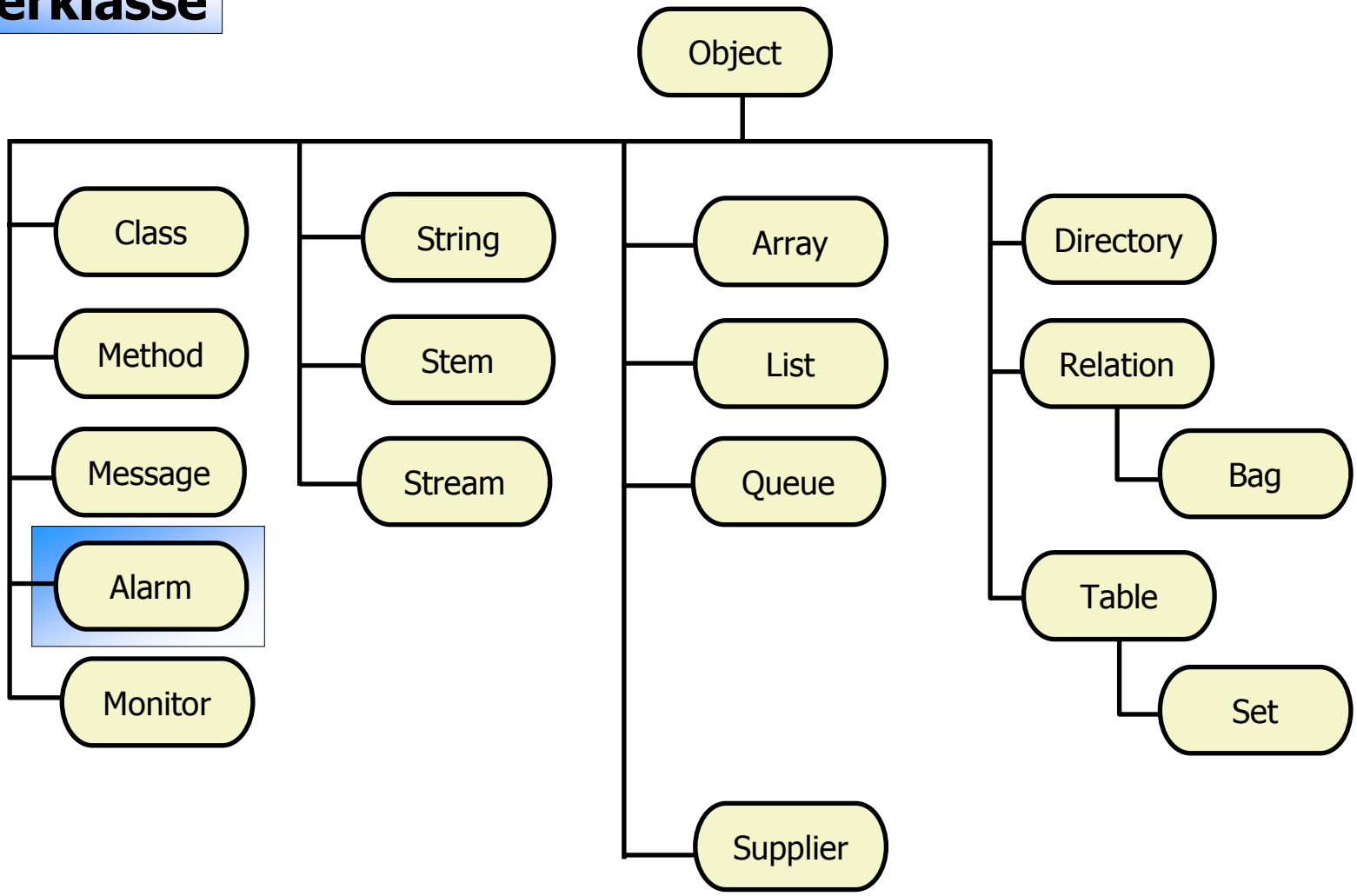
Fundamentale Klassen, 2

- Method
  - Interpreter erzeugt aufgrund der ::METHOD-Direktive Instanzen *(Methodenobjekte)* von diesem Typ
    - Beispiel: Methode SOURCE
      - Liefert die Methode im Quellcode (als Programmtext), sofern verfügbar
- Message
  - Interpreter erzeugt für jede Object Rexx-Nachricht zur Laufzeit Instanzen *(Nachrichtenobjekte)* von diesem Typ
    - Beispiel: Methode SEND
      - Schickt die Nachricht an das angegebene Objekt ab

© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum von Object Rexx, 3

Weckerklasse



© Prof. Dr. Rony G. Flatscher

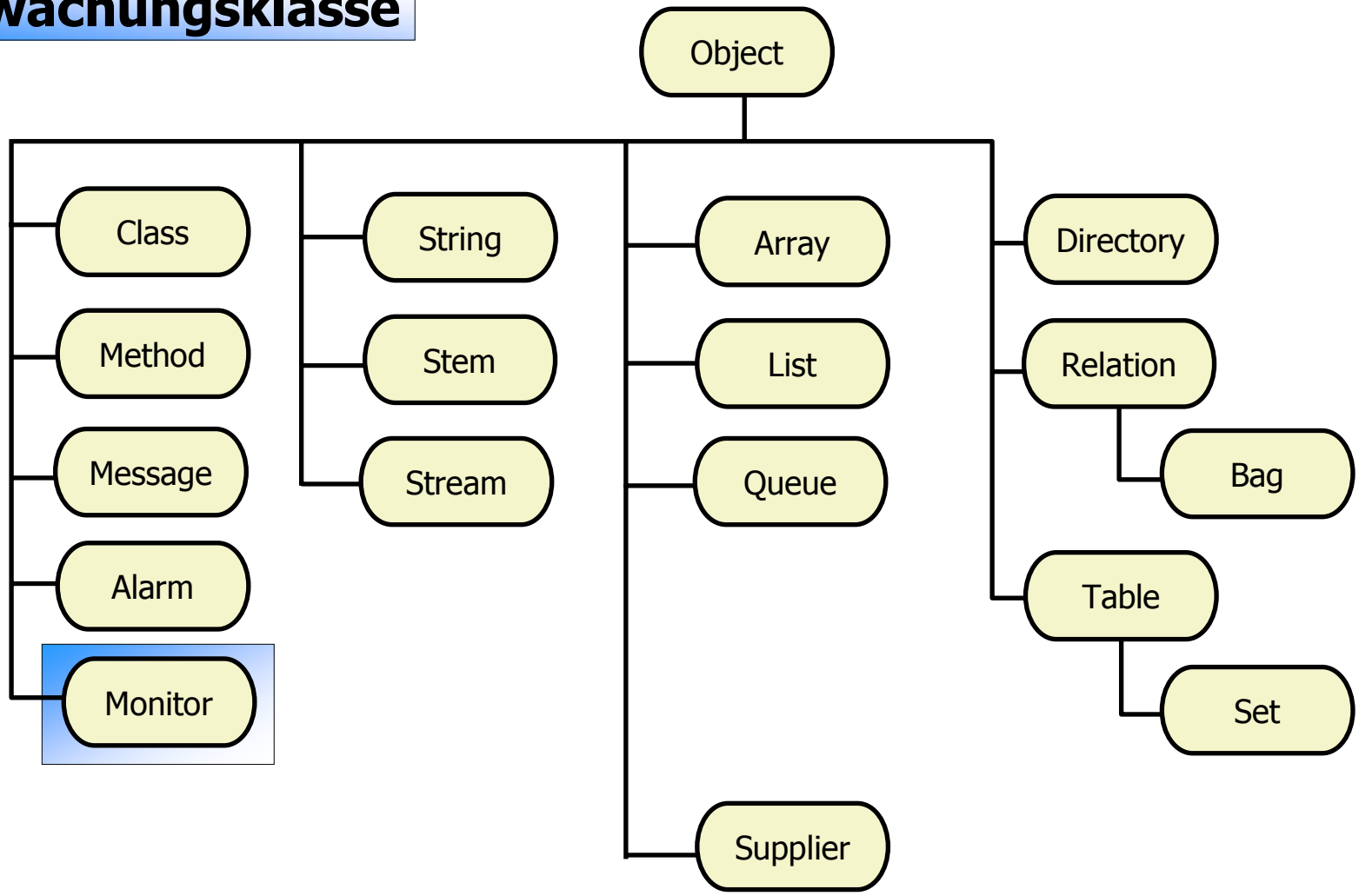
Weckerklasse

- Alarm
  - Alarmobjekte *(Weckerobjekte)* erlauben das Aktivieren von Nachrichten zu einem späteren Zeitpunkt
    - Derart aktivierte Nachrichten sind nebenläufig
    - Aktivierung kann festgelegt werden
      - In Stunden, Minuten, Sekunden von der Initialisierung an
      - Als gültiges Datum mit Uhrzeit
    - Beispiel: Methode CANCEL
      - Annuliert ein Weckerobjekt

© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum von Object Rexx, 4

Überwachungsklasse



© Prof. Dr. Rony G. Flatscher

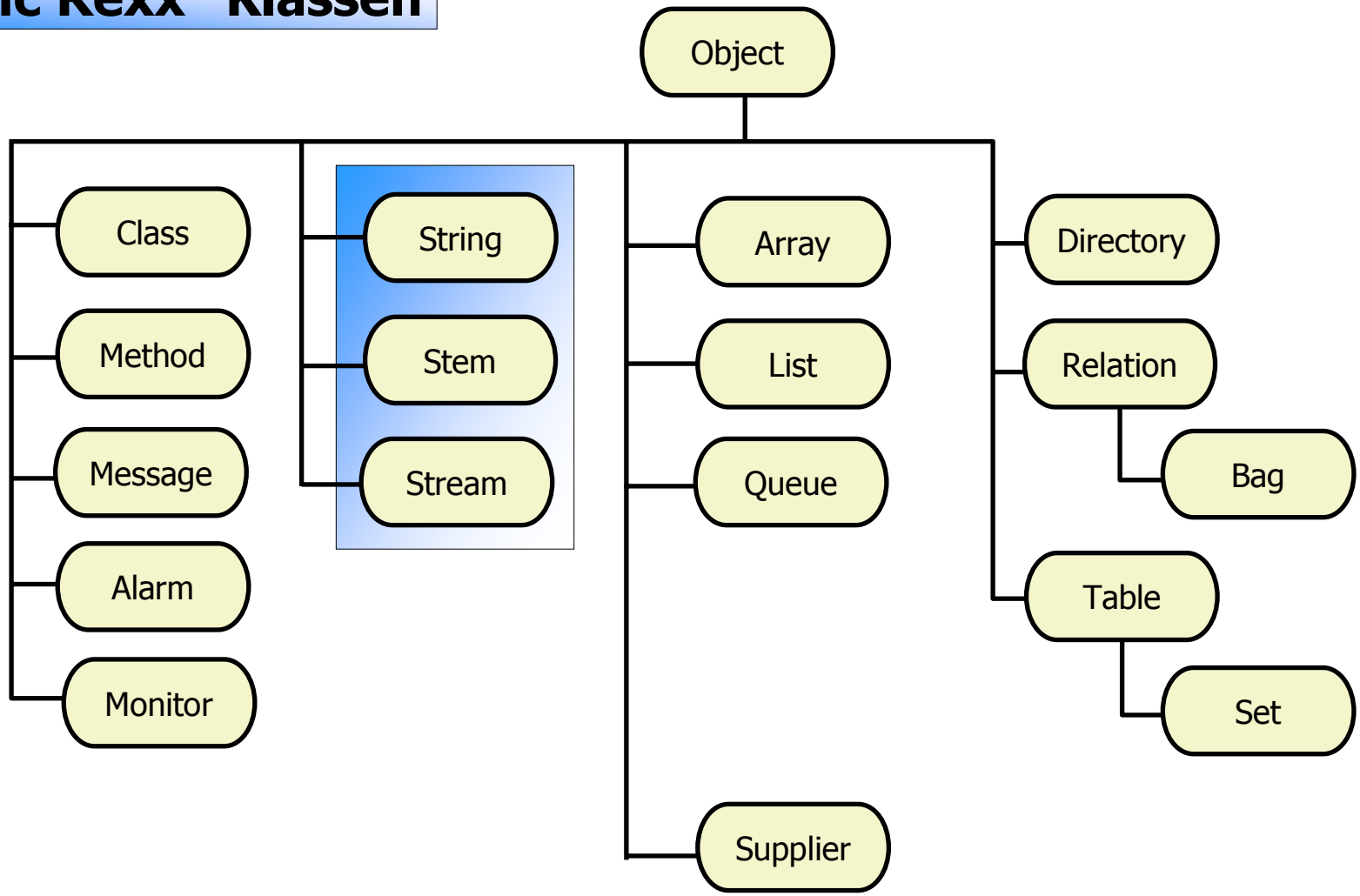
Überwachungsklasse

- Monitor
  - Monitorobjekte *(Überwachungsobjekte)* erlauben das Überwachen von Nachrichten, die dem angegebenen Objekt gesendet werden
    - Beispiel: Methode DESTINATION
      - Legt das zu überwachende Objekt fest beziehungsweise
      - Liefert das gerade überwachte Objekt zurück

© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum von Object Rexx, 5

"Classic Rexx" Klassen



© Prof. Dr. Rony G. Flatscher

"Classic Rexx" Klassen, 1

- String (1)
  - Stringobjekte *(Zeichenkettenobjekte)* verfügen über zahlreiche Methoden, die sämtliche zeichenketten- bezogenen Funktionen nachbilden
    - **Besonderheit:** Stringobjekte verändern niemals ihren Wert!
      - Dadurch wird gewährleistet, daß sich die Zeichenketten- objekte unter Object Rexx so verhalten, wie unter Classic Rexx

```
a = .string~new("hallo") /* neues Stringobjekt          */
a = "hallo"    /* neues Stringobjekt mit Wert "hallo" */
a = "aloha"    /* neues Stringobjekt mit Wert "aloha" */
a = "aloha"    /* neues Stringobjekt mit Wert "aloha" */

a = "a" || "b" /* neues Stringobjekt mit Wert "ab"    */
a = a   || "b" /* neues Stringobjekt mit Wert "abb"   */

a = 1 + 3      /* neues Stringobjekt mit Wert "4"     */
a = a + 3      /* neues Stringobjekt mit Wert "7"     */
```


© Prof. Dr. Rony G. Flatscher

"Classic Rexx" Klassen, 2

- String (2)
  - Zeichenkettenbezogene Funktionen werden vom Interpreter in ihre objektorientierte Form umgesetzt, indem die entsprechenden Methoden dem Zeichenkettenobjekt geschickt werden
  - Beispiel: Methode REVERSE
    - Dreht eine Zeichenkette um

```
SAY REVERSE("d:\pfad\datei.typ") /* Funktion */
SAY "d:\pfad\datei.typ"~REVERSE  /* Methode  */
```

Ausgabe:

```
pyt.ietad\dafp\:d
pyt.ietad\dafp\:d
```


© Prof. Dr. Rony G. Flatscher

"Classic Rexx" Klassen, 3

- Stem (1)
  - Stemobjekte erlauben beliebige Zeichenkettenindices
    - Der Stamm (englisch: "stem") des Bezeichners wird durch einen Punkt begrenzt

```
a.2 = "Ich bin a.2"
SAY a.1.b "/und\" a.2
```

Ausgabe:

```
A.1.B /und\ Ich bin a.2
```

```
a.  = "kein Eintrag"
a.2 = "Ich bin a.2"
SAY a.1.b "/und\" a.2
```

Ausgabe:

```
kein Eintrag /und\ Ich bin a.2
```

```
a = .stem~new("kein Eintrag") /* neues Stemobjekt */
a[2] = "Ich bin a.2"
SAY a[a.1.b] "/und\" a[2]
```

Ausgabe:

```
kein EintragA.1.B /und\ Ich bin a.2
```


© Prof. Dr. Rony G. Flatscher

"Classic Rexx" Klassen, 4

- Stem (2)
  - Stemobjekte *(Stammobjekte)* erlauben das Sammeln von beliebigen Objekten mit Hilfe Zeichenkettenindices
    - Beispiele: Methoden [] und []=

```
DROP a a. b b. /* Sicherstellen: Variablen gelöscht */
a = .stem~new("xyz")
a["holladi"] = "Eintrag für 'holla.di'"
b. = a /* Zwei Referenzen auf dasselbe Stemobjekt!  */
b.di.di.dumm = "Eintrag für 'DI.DI.DUMM'"
SAY "1:" a["holladi"]      "/und\" a~"[]"("DI.DI.DUMM")
tmp1 = "holladi"; tmp2 = "DI.DI.DUMM"
SAY "2:" a.tmp1            "/und\" a.[tmp2]
SAY "3:" b.tmp1            "/und\" b.[tmp2]
SAY "4:" a a. a.Unbekannt b b. b.Unbekannt a[Unbekannt]
```

Ausgabe:

```
1: Eintrag für 'holla.di' /und\ Eintrag für 'DI.DI.DUMM'
2: A.holladi /und\ A.DI.DI.DUMM
3: Eintrag für 'holla.di' /und\ Eintrag für 'DI.DI.DUMM'
4: xyz A. A.UNBEKANNT B xyz xyzUNBEKANNT xyzUNBEKANNT
```


© Prof. Dr. Rony G. Flatscher

"Classic Rexx" Klassen, 5

- Stream
  - Streamobjekte erlauben die Abarbeitung von Dateien
    - Beispiel: Methode NEW

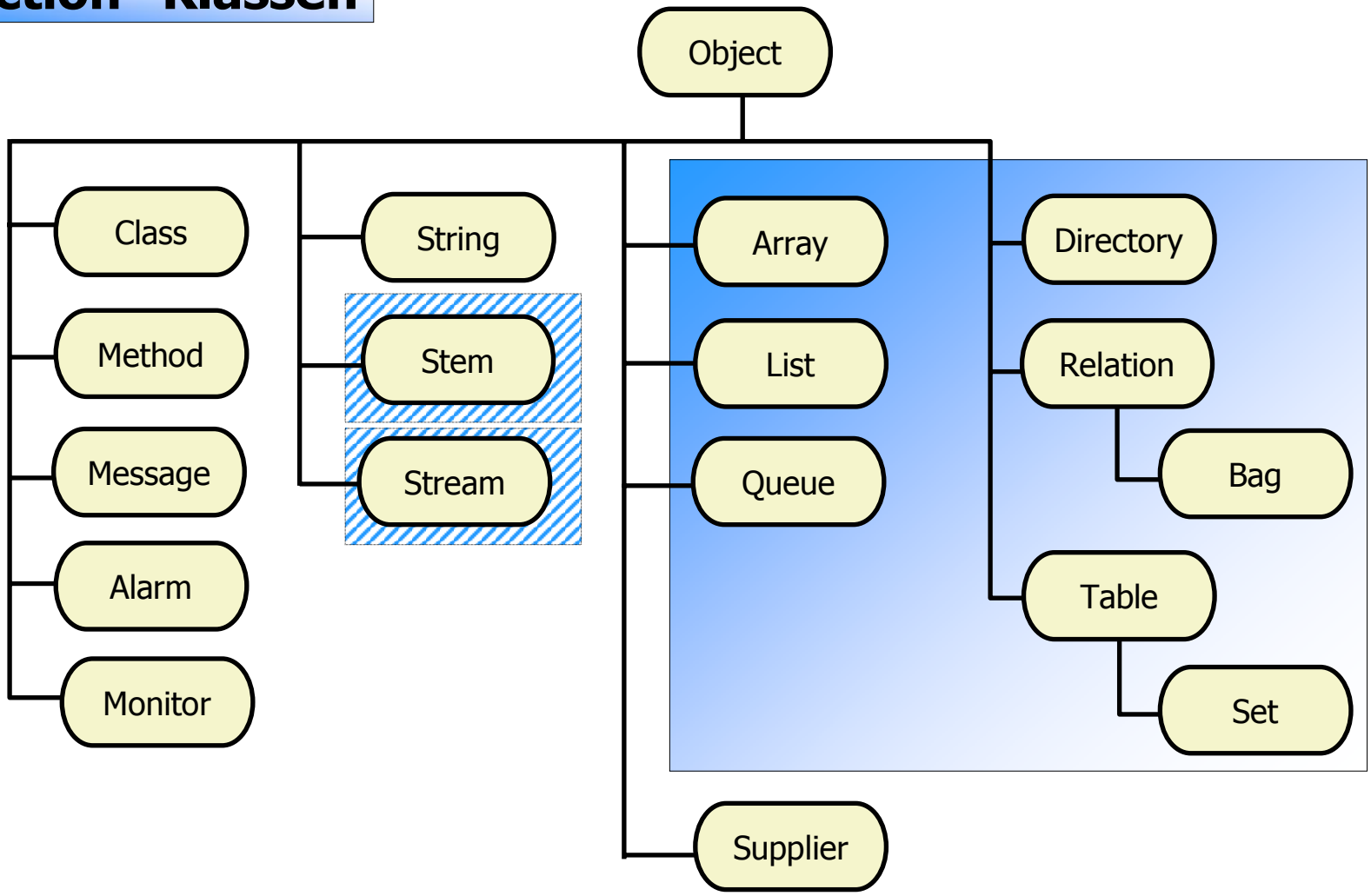
```
o = .stream ~NEW("test.dat")
```

      - Erlaubt das Abarbeiten der Datei **test.dat**, indem dem Streamobjekt **o** die entsprechenden Methoden gesendet werden, z.B. OPEN zum Öffnen, LINEIN (CHARIN) zum Einlesen, LINEOUT (CHAROUT) zum Schreiben, CLOSE zum Schließen

© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum von Object Rexx, 6a

"Collection" Klassen



© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 1a

- "Collection" Klassen *(Sammlungklassen)* erlauben das Speichern von *beliebigen* Object Rexx-Objekten
- Objekte werden in der Regel mit den Methoden
  - PUT beziehungsweise "[ ]=" **gespeichert** (gesammelt)

```
sammelobjekt ~PUT(objekt,index)
sammelobjekt ~"[]="(objekt,index) oder
sammelobjekt[index] = objekt
```

  - AT beziehungsweise "[ ]" **abgerufen**

```
sammelobjekt ~AT(index)
sammelobjekt ~"[]"(index) oder
sammelobjekt[index]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 1b

- Instanzen von "Collection" Klassen können statt mit NEW auch mit OF erzeugt werden, wobei der Methode OF eine Liste von zu sammelnden Objekte als Argument mitgegeben wird
- Die gesammelten Objekte können mit einem DO...OVER-Block einzeln abgearbeitet werden
  - Weitere Möglichkeit: SUPPLIER-Objekte (siehe weiter unten)
  - Abarbeitungsschleife

```
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 2

- "Collection" Klassen unterscheiden sich in
  - "Collection" Klassen ***ohne*** *benutzerdefinierten* Index
  - "Collection" Klassen ***mit*** *benutzerdefinierten* Index
- Klassen ohne benutzerdefinierten Index
  - Array
  - List
  - Queue
  - (Stream)

© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 3

- Array (1)
  - Arrayobjekte *(Feldobjekte)* erlauben das Speichern von Objekten mit vordefinierten numerischen Indizes, die mit 1 beginnen

```
tmpColl = .array ~of("a", "b", "b")
tmpColl[4] = "c"

SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Ausgabe:

```
an Array:
[a]
[b]
[b]
[c]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 4

- Array (2)
  - Arrayobjekte können auch über beliebig viele Dimensionen aufweisen
    - Achtung! Speicherbedarf ist das kartesische Produkt aller maximalen Einträge aller Dimensionen

```
tmpColl = .array ~new
tmpColl[2,3] = "a"
tmpColl ~"[]="("b", 1, 1)
tmpColl ~~put("b", 4, 5) ~~put("c", 1, 2)

SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Ausgabe:

```
an Array:
[b]
[c]
[a]
[b]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 5

- List
  - Listenobjekte erlauben das Speichern von Objekten in Form einer Liste, das heißt in einer festlegbaren Reihenfolge

```
tmpColl = .list ~of("a", "b", "b", "c")

SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Ausgabe:

```
a List:
[a]
[b]
[b]
[c]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 6

- Queue
  - Queueobjekte *(Schlangenobjekte)* erlauben das Speichern von Objekten am "Kopf" (PUSH) oder am "Hintern" (QUEUE), das heißt in einer festlegbaren Reihenfolge

```
tmpColl = .queue ~new
tmpColl ~~queue("a") ~~queue("b") ~~push("b") ~push("c")

SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Ausgabe:

```
a Queue:
[c]
[b]
[a]
[b]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 7

- Stream
  - Streamobjekte erlauben das Abarbeiten von "Zeichenströmen", hauptsächlich von Dateien

    o = .stream~NEW("test.dat")

    - Mit Hilfe des Streamobjekts **o** kann die Datei "**test.dat**" abgearbeitet werden, indem die entsprechenden Nachrichten geschickt werden, zum Beispiel: *OPEN*, *LINEIN (CHARIN)*, *LINEOUT (CHAROUT)*, *CLOSE*...

```
tmpColl = .stream ~new("test.dat")~~open
SAY "a" tmpColl~class~id || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
tmpColl~close
```

Mögliche Ausgabe:

```
a Stream:
[Das ist die erste Zeile.]
[]
[Die vorhergehende war leer, jetzt ist die dritte dran!]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 8

- Klassen mit benutzerdefinierten Index
  - Directory - pro Index nur *ein* Objekt assoziert
  - Relation - pro Index *beliebig viele* Objekte assoziert (ALLAT)
    - Bag - Index = assoziertes Objekt
  - Table - pro Index nur *ein* Objekt assoziert
    - Set - Index = assoziertes Objekt
  - (Stem - pro Index nur *ein* Objekt assoziert)
- Objekte werden in keiner bestimmten Reihenfolge in der Sammlung abgelegt
  - DO...OVER (und auch SUPPLIER-Objekte) liefern daher die gesammelten Objekte in einer beliebigen (unvorhersehbaren) Reihenfolge!

© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 9

- Directory
  - Directoryobjekte (*Verzeichnisobjekte*) erlauben das Speichern von Objekten mit benutzerdefinierten **Zeichenketten** als Indices

```
tmpColl = .directory ~new
tmpColl["a_index"] = "a"
tmpColl ~"[]="("b", "b_index")
tmpColl ~~PUT("b", "b_index" ) ~~PUT("c", "c_index")
tmpColl ~~wu  = "WU Wien"
tmpColl ~~rgf = "Rony G. Flatscher"
SAY "Abkürzung 'WU':" tmpColl~wu || ", 'RGF':" tmpColl~rgf
SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Mögliche Ausgabe:

```
Abkürzung 'WU': WU Wien, 'RGF': Rony G. Flatscher
a Directory:
[b_index]
[c_index]
[WU]
[RGF]
[a_index]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 10

- Relation
  - Relationsobjekte erlauben das Speichern von Objekten mit benutzerdefinierten **Objekten** als Indices

```
tmpColl = .relation ~new
tmpColl["a_index"] = "a"
tmpColl ~"[]="("b", "b_index")
tmpColl ~~PUT("b", "b_index" ) ~~PUT("c", "c_index")

SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Mögliche Ausgabe:

```
a Relation:
[b_index]
[c_index]
[a_index]
[b_index]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 11

- Bag
  - Bagobjekte (*Sackobjekte*) erlauben das mehrfache Speichern von Objekten, wobei für den Index das zu speichernde Objekt selbst herangezogen wird (und daher auch entfallen darf)

```
tmpColl = .bag ~new
tmpColl["a"] = "a"
tmpColl ~"[]="("b", "b")
tmpColl ~~PUT("b") ~~PUT("c")

SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Mögliche Ausgabe:

```
a Bag:
[a]
[b]
[c]
[b]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 12

- Table
  - Tableobjekte (*Tabellenobjekte*) erlauben das Speichern von Objekten mit benutzerdefinierten **Objekten** als Indices

```
tmpColl = .table ~new
tmpColl["a_index"] = "a"
tmpColl ~"[]="("b", "b_index")
tmpColl ~~PUT("b", "b_index" ) ~~PUT("c", "c_index")

SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Mögliche Ausgabe:

```
a Table:
[b_index]
[c_index]
[a_index]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 13

- Set
  - Setobjekte (*Mengenobjekte*) erlauben das einfache Speichern von Objekten, wobei für den Index das zu speichernde Objekt selbst herangezogen wird (und daher auch entfallen darf)

```
tmpColl = .set ~new
tmpColl["a"] = "a"
tmpColl ~"[]="("b", "b")
tmpColl ~~PUT("b") ~~PUT("c")

SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Mögliche Ausgabe:

```
a Set:
[a]
[b]
[c]
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 14

- Stem
  - Stemobjekte (*Stammobjekte*) erlauben das Speichern von Objekten mit benutzerdefinierten **Zeichenketten** als Indices

```
tmpColl = .stem ~new
tmpColl["a_index"] = "a"
tmpColl ~"[]="("b", "b_index")
tmpColl ~~PUT("b", "b_index" ) ~~PUT("c", "c_index")

SAY "a" tmpColl~class~id || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

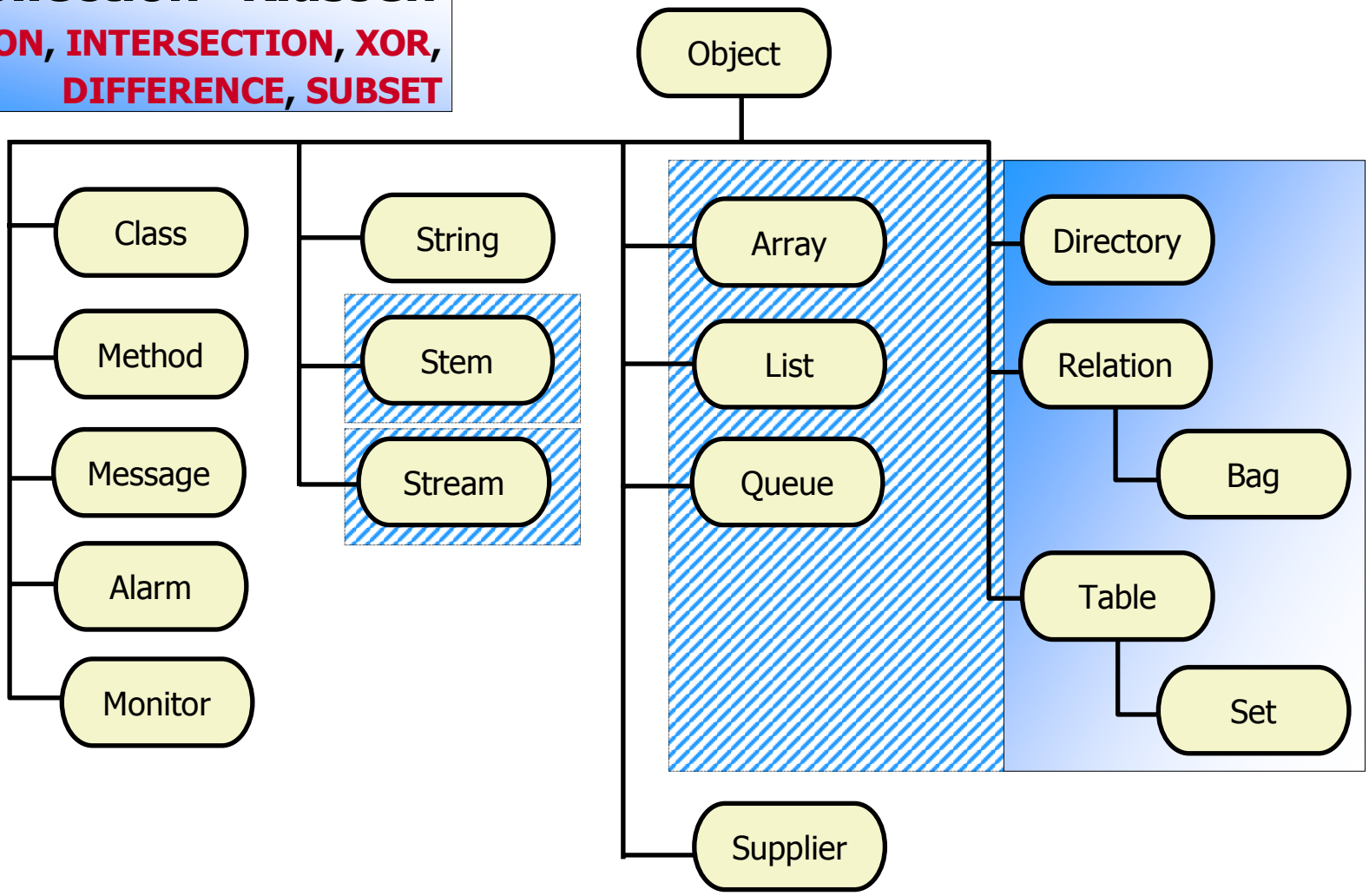
Mögliche Ausgabe:

```
a Stem
[b_index]
[c_index]
[a_index]
```


© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum von Object Rexx, 6b

"Collection" Klassen
mit UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET



© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 1
(UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET)

- Beispiel 1 (zwei Bags)

```
coll_1 = .bag ~of("a", "b", "b")
coll_2 = .bag ~of(     "b", "b", "c")
CALL dump coll_1~UNION(coll_2),          "UNION       "
CALL dump coll_1~INTERSECTION(coll_2), "INTERSECTION"
CALL dump coll_1~XOR(coll_2),            "XOR         "
CALL dump coll_1~DIFFERENCE(coll_2),   "DIFFERENCE  "
SAY coll_1~SUBSET(coll_1) "-" coll_1~SUBSET(coll_2)

::ROUTINE dump
  USE ARG tmpColl, title
  .stdout~CHAROUT( title tmpColl~string || ": ")
  DO item OVER tmpColl
     .stdout~CHAROUT("[" || item || "] ")
  END
  SAY
```

Ausgabe:

```
UNION        a Bag: [a] [b] [c] [b] [b] [b]
INTERSECTION a Bag: [b] [b]
XOR          a Bag: [a] [c]
DIFFERENCE   a Bag: [a]
1 - 0
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 2
(UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET)

- Beispiel 2 (Set und Bag)

```
coll_1 = .set ~of("a", "b", "b")
coll_2 = .bag ~of(    "b", "b", "c")
CALL dump coll_1~UNION(coll_2),          "UNION        "
CALL dump coll_1~INTERSECTION(coll_2), "INTERSECTION"
CALL dump coll_1~XOR(coll_2),            "XOR          "
CALL dump coll_1~DIFFERENCE(coll_2),   "DIFFERENCE   "
SAY coll_1~SUBSET(coll_1) "-" coll_1~SUBSET(coll_2)

::ROUTINE dump
  USE ARG tmpColl, title
  .stdout~CHAROUT( title tmpColl~string || ": ")
  DO item OVER tmpColl
     .stdout~CHAROUT("[" || item || "] ")
  END
  SAY
```

Ausgabe:

```
UNION        a Set: [a] [b] [c]
INTERSECTION a Set: [b]
XOR          a Set: [a] [c]
DIFFERENCE   a Set: [a]
1 - 0
```


© Prof. Dr. Rony G. Flatscher

"Collection" Klassen, 3
(UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET)

- Beispiel 3 (Bag und Set)

```
coll_1 = .bag ~of("a", "b", "b")
coll_2 = .set ~of(     "b", "b", "c")
CALL dump coll_1~UNION(coll_2),          "UNION        "
CALL dump coll_1~INTERSECTION(coll_2), "INTERSECTION"
CALL dump coll_1~XOR(coll_2),            "XOR          "
CALL dump coll_1~DIFFERENCE(coll_2),   "DIFFERENCE   "
SAY coll_1~SUBSET(coll_1) "-" coll_1~SUBSET(coll_2)

::ROUTINE dump
  USE ARG tmpColl, title
  .stdout~CHAROUT( title tmpColl~string || ": ")
  DO item OVER tmpColl
     .stdout~CHAROUT("[" || item || "] ")
  END
  SAY
```

Ausgabe:

```
UNION        a Bag: [a] [b] [c] [b] [b]
INTERSECTION a Bag: [b]
XOR          a Bag: [a] [b] [c]
DIFFERENCE   a Bag: [a] [b]
1 - 0
```


© Prof. Dr. Rony G. Flatscher

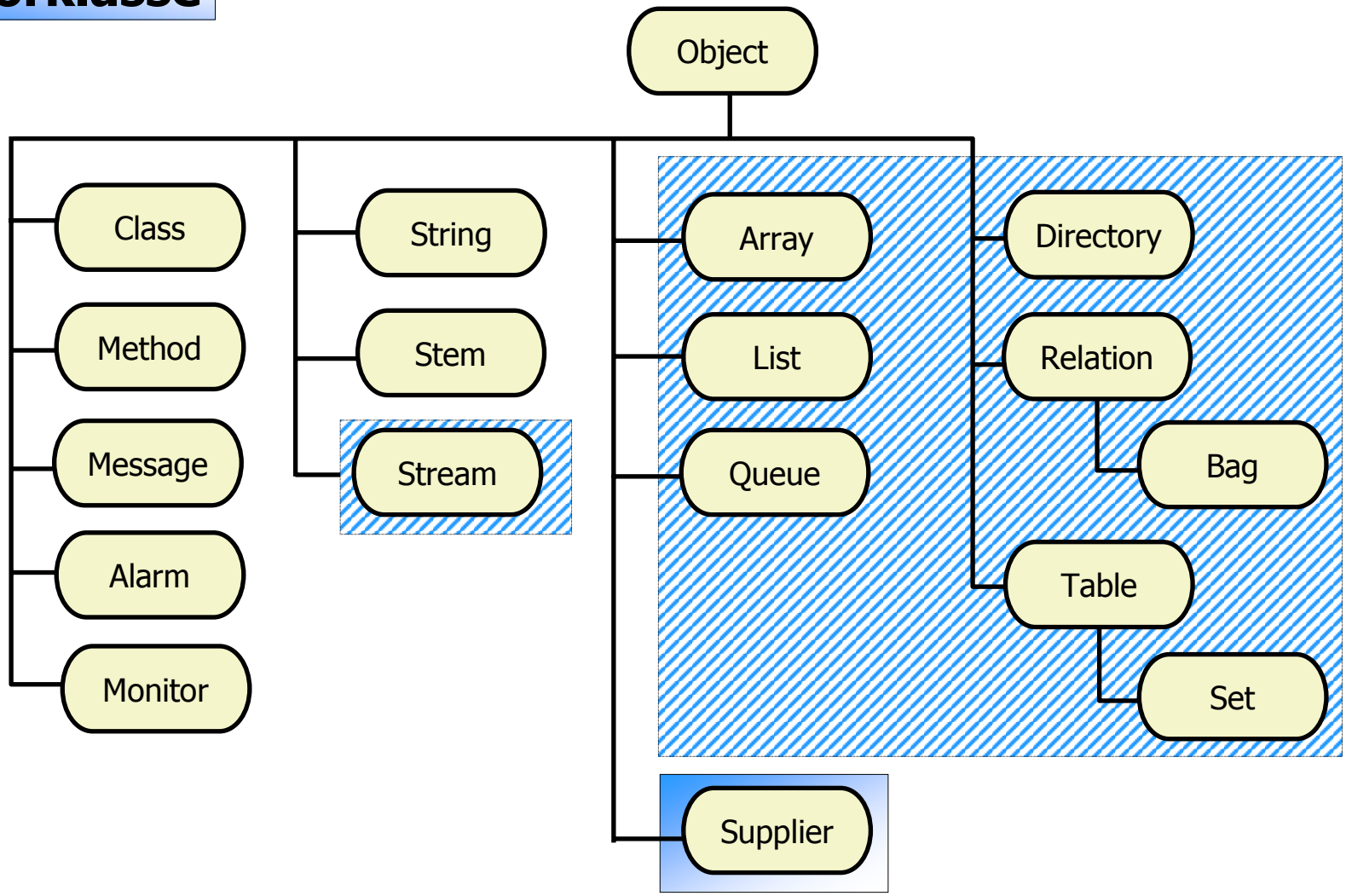
"Collection" Klassen, 4
(UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET)

- Ergebnis ist *immer* ein Ergebnisobjekt vom Typ des Empfänger-objektes
  - Argument einer Mengenmethode kann ein beliebiges Sammelobjekt sein, das in den Typ des Empfängerobjektes umgewandelt wird
    - Argument kann bei Bedarf in einem Zwischenschritt in ein Bagobjekt umgewandelt werden, wenn es über keine benutzerdefinierten Indices verfügt (Array, List, Queue, Stem, Stream)
      - Bag enthält die gesammelten Objekte (“Items”)

© Prof. Dr. Rony G. Flatscher

Klassifikationsbaum von Object Rexx, 6c

Iteratorklasse



© Prof. Dr. Rony G. Flatscher

Iteratorklasse, 1

- Supplier
  - Supplierobjekte *(Iteratorobjekte)* erlauben das Abarbeiten aller in einem Sammlungsobjekt enthaltenen Objekte
  - "Collection" (Sammel-) Klassen enthalten dafür eine entsprechende SUPPLIER-Methode
  - Abarbeitungsschleife

```
tmpSupp = tmpColl~SUPPLIER
DO WHILE tmpSupp~AVAILABLE
   SAY "index [" || tmpSupp~INDEX || "]" ,
       "item [" || tmpSupp~ITEM  || "]"
   tmpSupp~NEXT
END
```


© Prof. Dr. Rony G. Flatscher

Iteratorklasse, 2

- Beispiel 1 (benutzerdefinierter Index: Relation)

```
tmpColl = .relation ~new
tmpColl["a_index"] = "a"
tmpColl ~"[]="("b", "b_index")
tmpColl ~~PUT("b", "b_index" ) ~~PUT("c", "c_index")

SAY tmpColl~string || ":"

tmpSupp = tmpColl~SUPPLIER
DO WHILE tmpSupp~AVAILABLE
   SAY "index [" || tmpSupp~INDEX || "]" ,
       "item [" || tmpSupp~ITEM  || "]"
   tmpSupp~NEXT
END
```

Ausgabe:

```
a Relation:
index [b_index] item [b]
index [c_index] item [c]
index [a_index] item [a]
index [b_index] item [b]
```


© Prof. Dr. Rony G. Flatscher

Iteratorklasse, 3

- Beispiel 2 (vordefinierter Index: 2-dimensionaler Array)

```
tmpColl = .array ~new
tmpColl[2,3] = "a"
tmpColl ~"[]="("b", 1, 1)
tmpColl ~~put("b", 4, 5) ~~put("c", 1, 2)

SAY tmpColl~string || ":"

tmpSupp = tmpColl~SUPPLIER
DO WHILE tmpSupp~AVAILABLE
   SAY "index [" || tmpSupp~INDEX || "]" ,
       "item [" || tmpSupp~ITEM  || "]"
   tmpSupp~NEXT
END
```

Ausgabe:

```
an Array:
index [1] item [b]
index [2] item [c]
index [8] item [a]
index [20] item [b]
```


© Prof. Dr. Rony G. Flatscher

Iteratorklasse, 4

- Beispiel 3 (benutzerdefinierter Index: Set)

```
tmpColl = .set ~new
tmpColl["a"] = "a"
tmpColl ~"[]="("b", "b")
tmpColl ~~PUT("b") ~~PUT("c")

SAY tmpColl~string || ":"

tmpSupp = tmpColl~SUPPLIER
DO WHILE tmpSupp~AVAILABLE
   SAY "index [" || tmpSupp~INDEX || "]" ,
       "item [" || tmpSupp~ITEM  || "]"
   tmpSupp~NEXT
END
```

Ausgabe:

```
a Set:
index [a] item [a]
index [b] item [b]
index [c] item [c]
```


© Prof. Dr. Rony G. Flatscher