

# Open Object Rexx Tutorial

USE ARG, Routines, Abstract Datatype, Classes,  
Methods, Attributes, Messages, Scopes,  
Generalizing Class Hierarchy, Inheritance

**Prof. Rony G. Flatscher**

# Variables (Object Rexx)

- Variables are **References** to instances of Object Rexx classes
  - Strings
  - Stems
  - ... (more later ...)
- Arguments for routines (procedures/functions)
  - **PARSE ARG** statement
    - **Only** Strings allowed
    - No Stem-Variable !
    - **EXPOSE** statement allows access to a stem variable defined in the caller from an *internal* routine
  - **USE ARG** statement
    - **All** Objects are allowed as arguments

# Routine Directive (Object Rexx)

- Routine directives
  - Start with a double-colon (::)
  - Routines (both, defined internally or as directives) represent procedures and functions (= returning a value)
    - There is no **EXPOSE** statement available to the routine
  - After a successful syntax check they are made available in the scope
    - Of the program itself, and
    - In addition in all superordinate (calling) programs, *if* the keyword **PUBLIC** is given
  - Routine directives their **own scope**, as if they were a program of their own!
    - Therefore labels, i.e. *internal routines*, are available **within** routine directives

# Routine Directive (Object Rexx): 1a

```
/**/  
SAY pp("hello")  
CALL oha          /* routine is called */  
SAY pp("hello")  
  
EXIT 0  
pp : RETURN "<<<" || ARG(1) || ">>>"  
  
:: ROUTINE oha PUBLIC  
   SAY pp("holla")  
   EXIT 0  
   pp : RETURN "[" || ARG(1) || "]"
```

## Output:

```
<<<hello>>>  
[holla]  
<<<hello>>>
```

# Routine Directive (Object Rexx): 1b

```
/**/  
SAY pp("hello")  
CALL oha          /* routine is called */  
SAY pp("hello")  
  
EXIT 0  
pp : RETURN "<<<" || ARG(1) || ">>>"
```

```
:: ROUTINE oha PUBLIC  
SAY pp("holla")  
EXIT 0  
pp : RETURN "[" || ARG(1) || "]"
```

## Output:

```
<<<hello>>>  
[holla]  
<<<hello>>>
```

# Datatype (DT)

- Datatype
  - Defines the set of acceptable values
  - Defines the allowable operations (e.g. adding, concatenating)
  - Example
    - Datatype **Birthday**
      - E.g. defines a valid date and a valid time
      - Allowable operations, e.g. change/query the values of the stored date and time
    - Datatype **Person**
      - E.g. defines first name, family name, salary
      - Allowable operations, e.g. changing the values for first name, family name, salary, increase salary

# Datatype (DT)

## Classic REXX, Problems

- No means to *explicitly* define structures to represent a datatype
- No means to *explicitly* define operations which are only valid for a *specific* datatype
- Attempt to encode the structure with the help of
  - Strings
  - Stem-Variables

# Datatype (DT)

## Classic Rexx, Possible Solution 1

- Encoding with the help of **Strings**
  - E.g. data of type *Birthday*

```
"20050901 16:00"  
"20080229 19:19"
```
  - E.g. data of type *Person*

```
"Albert Einstein 45000"  
"Vera Withanyname 25000"
```
  - Processing only possible if the following is known to everyone
    - **Number** and **sequence** of the DT-"fields" (columns)
    - **Dimension** of the columns (variable, fixed width)
    - For instance encoded ASCII-files
      - Variable column width, hence a delimiting character necessary
        - E.g. "Comma Delimited Format"
      - Fixed column width



# Datatype (DT)

## Classic Rexx, Possible Solution 2

- Encoding with the help of **stems**

- E.g. data of type *Birthday*

- Collection of string encoded data with the help of stems

```
birth.1 = "20320901 16:00"
```

```
birth.2 = "20360229 19:19"
```

- Processing only possible if one knows the **number**, **sequence** and **width** of columns of the DT-"fields", e.g. `SysFileTree()`

- **Structuring** and collection of the string encoded data with the help of stems

```
birth.1.eDate = "20320901"
```

```
birth.1.eTime = "16:00"
```

```
birth.2.eDate = "20360229"
```

```
birth.2.eTime = "19:19"
```

- Processing already possible, if one knows **only** the identifiers (names) of the individual DT-"fields"!

# Datatype (DT)

## Classic Rexx, Possible Solution 3

- Encoding with the help of **stems**

- E.g. data of type *Person*

- **Structuring** with the help of stems

```
pers.eFirstName = "Albert"  
pers.eLastName  = "Einstein"  
pers.eSalary    = "45000"
```

and

```
pers.eFirstName = "Vera"  
pers.eLastName  = "Withanyname"  
pers.eSalary    = "25000"
```

- If using stems one **must** introduce an additional index in order to be able to store both persons above, independent of each other!
- The latter assignments ("Vera") would replace ("overwrite") the first ones ("Albert")

# Datatype (DT)

## Classic Rexx, Discussion of Possible Solutions

- DT structure
  - Encoding in strings and stems
    - Crook, as implementation dependent!
    - Error prone
- DT operations
  - No possibility to define operations for datatypes!
    - Internal routines (Functions or procedures) must be defined on their own
    - Direct access to strings and stems **must** be realized via **EXPOSE** statements
      - Problems with scopes, source of errors
- Insulating ("Encapsulating") of individual DT extensions ("instances") not possible

# Abstract Datatype (ADT)

- Abstract Datatype
  - **Schema** for the implementation of datatypes
    - Definition of **Attributes**
      - Results in the data structure
    - Definition of **Operations** ("Behaviour")
      - Method routines (Functions, Procedures)
  - Internal datastructures and values are usually
    - Not visible from the "outside"
    - Not directly editable from the "outside"
    - **Encapsulation !**
  - **Schema** must be implemented in an *appropriate* Programming language
    - Classic Rexx is not really *appropriate* for this
    - Object Rexx *is* - as any other object-oriented - programming language appropriate

# Abstract Datatype (ADT)

## Implementation with Object Rexx

- Abstract Datatype
  - **Schema** for the implementation of datatypes
    - **::CLASS** directive
      - Definition of **attributes** and therefore the internal datastructure
        - **EXPOSE** statement **within** methods or
        - **::METHOD** directive with the keyword **ATTRIBUTE**
      - Definition of **operations** (routines)
        - **::METHOD** directive
  - Instance of classes ("object")
    - Individual, unambiguously distinguishable instantiations of the same type
    - Possesses all the same attributes (constitute the datastructure as defined in the class) and operations ("methods of the class")

# Abstract Datatype (ADT)

## Example: Definition of an ADT

- Object Rexx implementation of the ADT *Birthday*

```
/**/
```

```
::CLASS Birthday  
::METHOD date ATTRIBUTE  
::METHOD time ATTRIBUTE
```

- Object
  - Instance (extension) of an ADT, i.e., of a class
    - Uniquely distinguishible from other objects (even) of the same type
  - Creation: sending the message **NEW** to a class
    - Accessing the class via its environment symbol
      - Dot, immediately followed by the class identifier (name of the class), e.g.

```
object1 = .String~NEW("hallo") /* Object Rexx version */  
object2 = "hallo" /* classic Rexx version */
```

# Object Rexx Messages

- **Interaction** (activating of methods) **with objects** (instances) **exclusively** via messages, which are sent to objects
  - Names of messages are the names of the methods, that should be invoked
  - Message operator ("**twiddle**") is the tilde character: ~
    - E.g. "ABC"~REVERSE yields: CBA
  - "Cascading" messages, two twiddles: ~~
    - E.g. "ABC"~~REVERSE yields (**attention!**): ABC
    - Sent messages activate the respective methods of the receiving object, result is **always** the receiving object!
      - Therefore multiple messages intended for the same object can be "cascaded" one after the other
    - Execution of messages: left to right

# Abstract Datatype (ADT)

## Example: Using of an ADT

- Object Rexx implementation of the ADT *Birthday*

```
/**/  
g1 = .Birthday~New  
g1~Date= "20320901"  
g1~Time= "16:00"  
g2=.Birthday~New~~"Date=" ("20360229") ~~"Time=" ("19:19")  
SAY g1~date g2~date g1~time g2~time
```

```
::CLASS Birthday  
::METHOD date ATTRIBUTE  
::METHOD time ATTRIBUTE
```

### Output:

```
20320901 20360229 16:00 19:19
```



# Abstract Datatype (ADT)

## Example: Using of an ADT

- Object Rexx implementation of the ADT *Birthday*

```
/**/  
g1 = .Birthday~New  
g1~Date= "20320901"  
g1~Time= "16:00"  
g2=.Birthday~New~~"Date=" ("20360229") ~~"Time=" ("19:19")  
SAY g1~date g2~date g1~time g2~time
```

```
::CLASS Birthday  
::ATTRIBUTE date  
::ATTRIBUTE time
```

### Output:

```
20320901 20360229 16:00 19:19
```

# Scope (1)

- Scope
  - Determines the visibility of labels, variables, classes, routines, methods and attributes
- **"Standard Scope"**
  - Determines which labels are visible
    - Labels are only visible within a program (until the end of the program **or** until the first directive led in by a double colon **::**, whatever comes first)
    - Labels within of **::ROUTINE** and **::METHOD** directives are only visible within these directives

# Scope (2)

- **"Procedure Scope"**

- Determines, which variables of the caller are visible (accessible) from within the called *internal* routine (procedure/function)
  - Internal routines (labels), **without** a **PROCEDURE** statement
    - All variables of the calling part of the program are accessible
  - Internal routines (labels), followed by a **PROCEDURE** statement
    - Variables of the calling part of the program are **not** accessible (are hidden)
      - **"Local scope"**
      - **But:** with the help of the **EXPOSE** statement which may immediately follow a **PROCEDURE** statement one can deliberately define direct access to variables of the calling part of the program

# Scope (3)

- **"Program Scope"**
  - Determines that all classes and routines defined in a program are accessible
    - **Local classes** and **routines** cannot be hidden/overwritten
    - Classes and routines can be defined to be **public**
  - In addition, this scope determines, that *public classes* and *public routines* of called or required (**::REQUIRES** directive) programs become accessible
    - **Attention!**
      - If *different* programs are called one after the other, and contain *public classes* or *public routines* with the *same names*, then those classes/routines are accessible that are defined in the *last called program*

# Scope (4)

- **"::Routine Scope"**
  - Defines it own scope for
    - Labels ("standard scope") and
    - Variables ("procedure scope")
  - Accessing classes and routines is determined by the "program scope"

# Scope (5)

- **"::Method Scope"**
  - Defines its own scope for
    - Labels ("standard scope") and
    - Variables ("procedure scope")
  - Accessing classes and routines is determined by the "program scope"
  - Attributes
    - Within a method it is possible to use the **EXPOSE** statement (immediately following the method directive) to list those attributes of the class which should be made directly available for access from within the method.
    - Defining attributes and their access methods can be alternatively carried out by using an **ATTRIBUTE** method directive.

# Scope (6)

- **"::Method Scope"** (continued)
  - Determines *which* attributes can be accessed *directly* from within a method
  - There are two types of scopes which determine the accessibility of attributes
    - Attributes, which are defined in methods assigned to classes
      - Methods defined after a class directive
      - Share the same set of ("instance") attributes
    - Attributes, which are defined in "free running methods"
      - Methods which are defined *before* a class directive
      - Share the same set of ("free running") attributes
      - *Hint*: accessing free running methods is possible via the environment symbol **.METHODS** from within the program where there are defined

# Overview of Scopes

- Rexx und Object Rexx
  - Standard scope
    - Labels, variables
  - Procedure scope
    - Variables in *internal* routines (procedures/functions)
- Object Rexx
  - Program scope
    - Accessing local and public classes and routines of called/required programs
  - Routine scope
    - Standard+procedure+program scope
  - Method scope
    - Standard+procedure+program plus accessibility of attributes
      - Instance methods: methods, which are defined for a class ("instance" attributes)
      - Free running methods: methods, which are defined **before** any class directive ("free running" attributes)



# Abstract Datatype "Person"

## Implementation in Object Rexx, 1

```
/**/
```

```
p1 = .Person~New; p1~firstName= "Albert";
```

```
p1~familyName= "Einstein"; p1~salary=45000
```

```
p2=.Person~New~~"firstName=("Vera")~~"salary=("25000")
```

```
p2~~"familyName=("Withanyname")
```

```
SAY p1~firstName p1~familyName p1~salary
```

```
SAY p2~firstName p2~familyName p2~salary
```

```
SAY "Total costs of salaries:" p1~salary + p2~salary
```

```
::CLASS Person
```

```
::METHOD firstName ATTRIBUTE
```

```
::METHOD familyName ATTRIBUTE
```

```
::METHOD salary ATTRIBUTE
```

### Output:

```
Albert Einstein 45000
```

```
Vera Withanyname 25000
```

```
Total costs of salaries: 70000
```

# Abstract Datatype "Person"

## Implementation in Object Rexx, 2

```
/**/  
p1 = .Person~New; p1~firstName= "Albert";  
p1~familyName= "Einstein"; p1~salary= "45000"  
p2=.Person~New~~"firstName=" ("Vera")~~"salary=" (25000)  
p2~~"familyName=" ("Withanyname")  
SAY p1~firstName p1~familyName p1~salary p2~firstName  
SAY p1~firstName p1~salary p1~~increaseSalary (10000) ~salary
```

```
::CLASS Person  
::ATTRIBUTE firstName  
::ATTRIBUTE familyName  
::ATTRIBUTE salary  
::METHOD increaseSalary  
EXPOSE salary  
USE ARG increase  
salary = salary + increase
```

### Output:

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

# Creating Objects

- Creating new objects
  - The **NEW** message is sent to the class
  - Result is a reference to an object (an instance) of the class
- **If** there is a method with the name **INIT** defined for a class, then this method will be invoked, before control returns. This is realized by way of sending the message **INIT** to the newly created object from within the **NEW** method.
  - If the message **NEW** received arguments, these will be forwarded **in the same sequence** with the **INIT** message to the newly created object
- The **INIT** method is also called ***"constructor"***

# Abstract Datatype "Person"

## Implementation in Object Rexx, Constructor

```
/**/  
p1 = .Person~New("Albert", "Einstein", "45000")  
p2 = .Person~New("Vera", "Withanyname", 25000)  
SAY p1~firstName p1~familyName p1~salary p2~firstName  
SAY p1~firstName p1~salary p1~~increaseSalary(10000) ~salary  
::CLASS Person  
::METHOD INIT  
  EXPOSE firstName familyName salary  
  USE ARG firstName, familyName, salary  
::METHOD firstName ATTRIBUTE  
::METHOD familyName ATTRIBUTE  
::METHOD salary ATTRIBUTE  
::METHOD increaseSalary  
  EXPOSE salary  
  USE ARG increase  
  salary = salary + increase
```

### Output:

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

# Deleting of Objects

- Objects are automatically deleted from the runtime system, if they are not referenced anymore (becoming "garbage")
  - **If** there is a method named **UNINIT** defined for a class, then this method will be invoked, right before the unreferenced object gets deleted. This will be invoked by the runtime system by sending the object the message **UNINIT**.
- The **UNINIT** method is called "***destructor***"

# The Rexx "DROP" statement

- **DROP** statement

- The **DROP** statement allows the explicit deleting of a variable
- If a variable is destroyed its reference to an existing object is removed
  - There is still the possibility that there are other variables which still possess references to such an object

# Abstrakter Datentyp "Person"

## Umsetzung in Object Rexx, Destruktor

```
/**/  
p1 = .Person~New("Albert","Einstein","45000")  
p2 = .Person~New("Vera","Withanyname",25000)  
SAY p1~firstName p1~familyName p1~salary p2~firstName  
SAY p1~firstName p1~salary p1~~increaseSalary(10000)~salary  
DROP p1; DROP p2; CALL SysSleep( 15 ); SAY "Finish."  
  
::CLASS Person  
::METHOD INIT  
  EXPOSE firstName familyName salary  
  USE ARG firstName, familyName, salary  
  
::METHOD UNINIT  
  EXPOSE firstName familyName salary  
  SAY "Object: <\"firstName familyName salary\"> is about to be destroyed."  
  
::METHOD firstName ATTRIBUTE  
::METHOD familyName ATTRIBUTE  
::METHOD salary ATTRIBUTE  
::METHOD increaseSalary  
  EXPOSE salary  
  USE ARG increase  
  salary = salary + increase
```

### Output, for example:

```
Albert Einstein 45000 Vera  
Albert 45000 55000  
Object: <Vera Withanyname 25000> is about to be destroyed.  
Finish.  
Object: <Albert Einstein 55000> is about to be destroyed.
```

# Abstract Datatype (ADT)

## Implementation in Object Rexx

- Abstract Datatype (Repetition)
  - **Schema** for the implementation of datatypes
    - Definition of **Attributes**
      - Results in the data structure
    - Definition of **Operations** ("Behaviour")
      - Method routines (Functions, Procedures)
  - Internal datastructures and values are usually
    - Not visible from the "outside"
    - Not directly editable from the "outside"
    - **Encapsulation !**
  - **Schema** must be implemented in an *appropriate* Programming language
    - Classic Rexx is not really *appropriate* for this
    - Object Rexx *is* - as any other object-oriented - programming language appropriate



# Classification Tree (Generalization Hierarchy)

- Generalization Hierarchy, "Classification Tree"
  - Allows **classification of instances** (Objects), e.g. from biology
  - **Ordering of classes in superclasses and subclasses** (schemata)
    - Subordered classes ("subclasses") **inherits** all properties of all superclasses up to and including the root class
    - Subclasses **specialize** in one way or the other the superclass(es)
      - "Defining of differences"
  - Sometimes it may make sense, that a subclass specializes directly more than one superclass at the same time ("**multiple inheritance**")
    - Example: Classes representing landborn and waterborn animals, where there exists a class "amphibians", which inherits directly from the landborn and waterborn animals

# Object Rexx: Classification Tree, 1

- Prefabricated "class tree"
  - Root class of Object Rexx is named "Object"
  - All user defined classes are assumed to specialize the class "Object", if no superclass is explicitly given
  - Single and multiple inheritance possible

# Object Rexx: Classification Tree, 2

- Search order
  - Conceptually, the object receiving a message, starts searching for a method by the name of the received message and if found invokes it with the supplied arguments
  - If such a method is not found in the class, from which the object is created, then the search is continued in the direct superclass up to and including the root class
  - If the method is not even found in the root class "**Object**", then an error exception is thrown ("Object does not understand message")
    - If there is a method named **UNKNOWN** defined, then instead of creating an exception the runtime system will invoke that method, supplying the name of the unknown method and its arguments, if any were supplied with the message

# Object Rexx: Classification Tree, 2

- Search order (continued)
  - For the purpose of searching there are special, pre-set variables which are **only available from within methods**
    - **super**
      - Always contains a reference to the immediate superclass
      - Allows re-routing the starting class for searching for methods to the superclass
    - **self**
      - Always contains a reference to the object for which the method got invoked
      - This way it becomes possible to send messages to the object from within a method
  - **super** and **self** determine the class, where the search for methods starts which carry the same name as the message

# Example "Dog", 1

- Problem description
  - "Animal SIG" keeping dogs
    - Normal dogs
    - Little dogs
    - Big dogs
  - All dogs possess a name and are able to bark
    - Normal dogs bark "Wuff Wuff"
    - Little dogs bark "wuuf"
    - Big dogs bark "WUFFF! WUFFF!! WUFFF!!!"
  - Define appropriate classes taking advantage of inheritance (search order)

# Example "Dog", 2

- Definition of a class "**Dog**", which possess all properties which are common to all types of dogs

```
/**/  
h1 = .Dog ~NEW ~"NAME=" ("Sweety") ~Bark
```

```
:::CLASS Dog  
:::METHOD Name ATTRIBUTE  
:::METHOD Bark  
SAY self~Name": "Wuff Wuff"
```

## Output:

```
Sweety: Wuff Wuff
```

# Example "Dog", 3

- Definition of a class "**BigDog**", which possesses all properties common to all big dogs

```
/**/  
h1 = .Dog      ~NEW ~~"NAME=" ("SweetY")    ~~Bark  
      .BigDog  ~NEW ~~"NAME=" ("Grobian")    ~Bark  
::CLASS  Dog  SUBCLASS  Object  
::METHOD Name      ATTRIBUTE  
::METHOD Bark  
      SAY self~Name": " "Wuff Wuff"  
::CLASS  BigDog  SUBCLASS  dog  
::METHOD Bark  
      SAY self~Name": " "WUFFF! WUFFF!! WUFFF!!!"
```

## Output:

SweetY: Wuff Wuff

Grobian: WUFFF! WUFFF!! WUFFF!!!

# Example "Dog", 5

- Definition of a class "**LittleDog**", which possesses all properties common to all little dogs

```
/**/  
.Dog~NEW      ~~"NAME=" ("Sweety")  ~Bark  
.BigDog~NEW   ~~"NAME=" ("Grobian") ~Bark  
.LittleDog~NEW ~~"NAME=" ("Arnie")   ~Bark  
  
::CLASS Dog          SUBCLASS Object  
::METHOD Name        ATTRIBUTE  
::METHOD Bark  
  SAY self~Name:" "Wuff Wuff" "-" self  
  
::CLASS BigDog      SUBCLASS dog  
::METHOD Bark  
  SAY self~Name:" "WUFFF! WUFFF!! WUFFF!!!" "-" self  
  
::CLASS "LittleDog" SUBCLASS dog  
::METHOD Bark  
  SAY self~Name:" "wuuf" "-" self
```

## Output:

```
Sweety: Wuff Wuff - a DOG  
Grobian: WUFFF! WUFFF!! WUFFF!!! - a BIGDOG  
Arnie: wuuf - a LittleDog
```



# Multithreading

- Multithreading
  - Multiple parts of a program execute at the *same time* (in parallel)
  - Possible problems
    - Data integrity (Object integrity)
    - Deadlocks
- Object REXX
  - **Inter** Object-Multithreading
    - *Different* objects (even of one and the same class) are sheltered from each other and can be active at the same time
  - **Intra** Object-Multithreading
    - ***Within*** an instance (an object) multiple methods can execute at the same time, if they are defined in *different classes*