Department of Information Systems and Operations Management

Procedural and Object-oriented Programming 4

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

Business Programming 1 | **Business Programming 2**

REXX → **ooRexx** → **BSF4ooRexx**

Basics, Parsing	Commands, APIs	Window-Automatisation, Web-Scripting	Security, Debugging	Graphical User Interfaces (GUI), Sockets, ...

Prof. Rony G. Flatscher

Datatype (DT), 1

- What is a datatype?
  - Examples from school?
- What gets determined with a datatype?
- What is an abstract datatype (ADT)?

Prof. Rony G. Flatscher

Datatype (DT), 2

- A datatype determines
  - The allowable values
  - The operations that can be carried out with these values
- Abstract datatype (ADT)
  - A generalized description/definition ("schema") of a datatype
  - Defines the allowable values in general terms
  - Defines the operations that can be carried out in general terms

Datatype (DT), 3

- Examples
  - Datatype **Birthday**
    - E.g. defines a valid date and a valid time attribute (field), i.e. a structure
    - Allowable operations, e.g. change/query the values of the stored date and time, calculate differences, e.g. how many years have passed since the birthday
  - Datatype **Person**
    - E.g. defines first name, family name, salary attributes (fields), i.e. a structure
    - Allowable operations, e.g. changing/querying the values for first name, family name, salary, increase salary
- Problem in Classic Rexx
  - No means to explicitly define structures to represent and name a datatype
  - No means to explicitly define operations for a specific datatype only

Possible Solution 1: Encoding with Strings

- Encoding of data of type **Birthday**
  - `"20050901 16:00"`
  - `"20080229 19:19"`
- Encoding of 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 attributes (fields)
  - **Dimension** of the columns (variable, fixed width)
  - For instance encoded ASCII-files could be encoded as
    - Fixed column width: each attribute (field) starts at the same position (column)
    - Variable column width: a separating (delimiting) character necessary between attributes
      - E.g. a blank or using "comma-separated values (CSV)" to separate attribute (field) values

Possible Solution 2: Encoding with Stems

- Data of type **Birthday**
  - Encoding of data in a string, storing a collection with a stem (birth.)

```
birth.1 = "20320901 16:00"
birth.2 = "20360229 19:19"
```

    - Processing only possible if one knows the **number**, **sequence** and **width** of the values representing the DT attributes (fields), e.g. SysFileTree()
  - Structuring and collecting of data with a stem (birth.) and indexes (eDate, eTime)

```
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 (indexes) of the individual DT attributes (fields)!

Cf. rexxref.pdf (1.13.4. Stems)

Possible Solution 3: Encoding with Stems

- 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 (e.g. a number) in order to be able to store both persons above, independent of each other!
  - Otherwise the latter assignments (`"Vera"` …) would replace ("overwrite") the first ones (`"Albert"` …)!

Cf. rexxref.pdf (1.13.4. Stems)

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)

- Implementing an ADT schema with ooRexx
  - **::CLASS** directive
    - Definition of **attributes** (fields) and therefore the internal datastructure
      - **::ATTRIBUTE directive**
      - **EXPOSE** statement denoting attributes (fields) ***within*** method routines
    - Definition of **operations** (method routines)
      - **::METHOD** directive
- Instances ("values", "objects") of datatypes ("classes", "types")
  - Individual, unambiguously distinguishable instantiations of the same type
  - Possess all the same attributes (constitute the datastructure as defined in the class) and operations ("methods of the class")

Cf. rexxref.pdf (3.3. ::CLASS)

Definition of an Abstract Datatype (ADT)

- Object Rexx implementation of the ADT **Birthday**

```
/**/
::CLASS Birthday
::ATTRIBUTE date
::ATTRIBUTE time
```

- Object
  - Instance (extension, value) of an ADT, i.e., of a class
    - Uniquely distinguishable 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 */
```


Cf. rexxref.pdf (3.3. ::CLASS)

Object Rexx Messages, 1

- Conceptually, objects are regarded to be living things in ooRexx with which one communicates using messages! :)
  - If an object receives a message (with or without arguments) it
    - Searches for a method by the name of the received message in its class
      - If found, it invokes the method, supplying the received message arguments, if any, and returns any value the method may have returned
      - If not found the object searches the class hierarchy to find and invoke the method as described above
        - If there is no method found by the object it will raise a runtime condition with the error message `"Object does not understand message"` and the interpreter stops the execution of the program
  - A message term consists at least of the receiving object, the message operator (`~`) and the message name to be sent to the object, e.g.

```
object = .birthday~new
```


Cf. rexxref.pdf (1.11.4. Message Terms)

Object Rexx Messages, 2

- Interaction (activating of methods) with objects (instances, values) is *only* possible via messages
  - Names of messages are the names of the methods, that the object must find and invoke on behalf of the programmer
  - Message operator ("twiddle") is the tilde character: `~`
    - `"ABC"~REVERSE` yields: `"CBA"`
  - "Cascading" messages, two twiddles: `~~`
    - `"ABC"~~REVERSE` yields (*attention!*): `"ABC"`
    - Sent messages activate the respective methods of the receiving object, however, upon return the interpreter changes the result to be *always* the receiving object!
      - Therefore multiple messages intended for the same object can be "cascaded" one after the other ("cascading messages) by using two tildes (`~~`)
  - Execution (resolution) of messages: from left to right

Cf. rexxref.pdf (1.11.4. Message Terms)

Using of an Abstract Datatype (ADT), 1

- 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
```

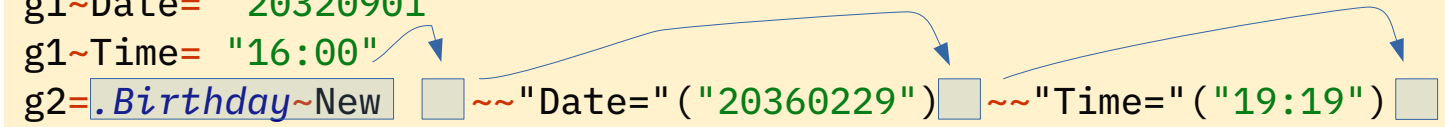

Cf. rexxref.pdf (1.11.4. Message Terms)

Using of an Abstract Datatype (ADT), 2

- 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
```


Cf. rexxref.pdf (1.11.4. Message Terms)

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**"

          However: with the help of the **EXPOSE** subkeyword on 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 after, 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 its 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"
  - Direct access to attributes
    - Within a method it is possible to use the **EXPOSE** statement (must be the first statement in the method routine) to list those attributes of the class which should be made directly available for access from within the method routine

Prof. Rony G. Flatscher

Scope, 6

- **"Method Scope" (continued)**
  - Determines which attributes can be accessed directly from within a method
  - There are two types of methods which determine the accessibility of attributes
    - Methods that are assigned to classes
      - Method and attribute directives defined after a class directive get assigned to that class
      - Expose and share the same set of instance/object attributes
    - "Floating methods" (advanced concept, can be used for dynamic programming)
      - Methods which are defined before a class directive are called "floating methods"
      - All floating methods can expose and share the same attributes with each other
      - Hint: accessing floating methods is possible via the environment symbol **.METHODS** from within the program where they 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
      - Methods assigned to a class: methods, which are defined for a class ("instance/object attributes")
      - Floating methods: methods, which are defined before any class directive ("floating attributes")

Abstract Datatype "Person", 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
::ATTRIBUTE firstName
::ATTRIBUTE familyName
::ATTRIBUTE salary
```

Output:

```
Albert Einstein 45000
Vera Withanyname 25000
Total costs of salaries: 70000
```

Abstract Datatype "Person", 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 (constructing) new objects (values, instances) can be done by sending the NEW message to a class
  - The NEW method will create the new object (instance, value) and will send it the message INIT to allow it to initialise
    - If the NEW message has arguments, they get forwarded with the INIT message in the same order
  - The NEW method returns the reference to the newly created object (instance, value) as its result
- Hence, if we define an INIT method for a class, we can use it to initialise an object immediately after it got created (constructed)
  - The INIT method is therefore also called "***constructor***"

Cf. rexxref.pdf (4.2.9. Initialization)

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

Output:

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


Cf. rexxref.pdf (4.2.9. Initialization)

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 by the garbage collector by sendig it the UNINIT message.
- The UNINIT method is therefore called "***destructor***"

Cf. rexxref.pdf (4.2.10. Object Destruction and Uninitialization)

The Rexx "DROP" statement

- **DROP** statement
  - The **DROP** statement allows the explicit deletion 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 that reference such an object

Cf. rexxref.pdf (2.5. DROP)

Destructor

```
/**/
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."
::ATTRIBUTE firstName
::ATTRIBUTE familyName
::ATTRIBUTE salary
::METHOD increaseSalary
  EXPOSE salary
  USE ARG increase
  salary = salary + increase
```

Output (maybe as uninit may run at different times):

```
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) - Roundup

- 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
  - 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, 1

- Generalization Hierarchy, "Classification Tree"
  - Allows **classification of instances** (Objects), e.g. from biology
  - **Ordering of classes in superclasses and subclasses** (schemata)
    - Subordered classes ("subclasses") **inherit** 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 landborne and waterborne animals, where there exists a class "amphibians", which inherits directly from the landborne and waterborne animals

Classification Tree, 2

- 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

Classification Tree: Search Order, 1

- 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 Object
- If the method is not even found in the root class Object, then an error condition gets 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

Classification Tree: Search Order, 2

- 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
::ATTRIBUTE Name
::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
::ATTRIBUTE Name
::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", 4

- 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
::ATTRIBUTE Name
::METHOD Bark
  SAY self~Name":" "Wuff Wuff"
::CLASS  BigDog    SUBCLASS dog
::METHOD Bark
  SAY self~Name":" "WUFFF! WUFFF!! WUFFF!!!"
::CLASS  LittleDog SUBCLASS dog
::METHOD Bark
  SAY self~Name":" "wuuf"
```

Output:

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

Example "Dog", 4

- 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
::ATTRIBUTE Name
::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*