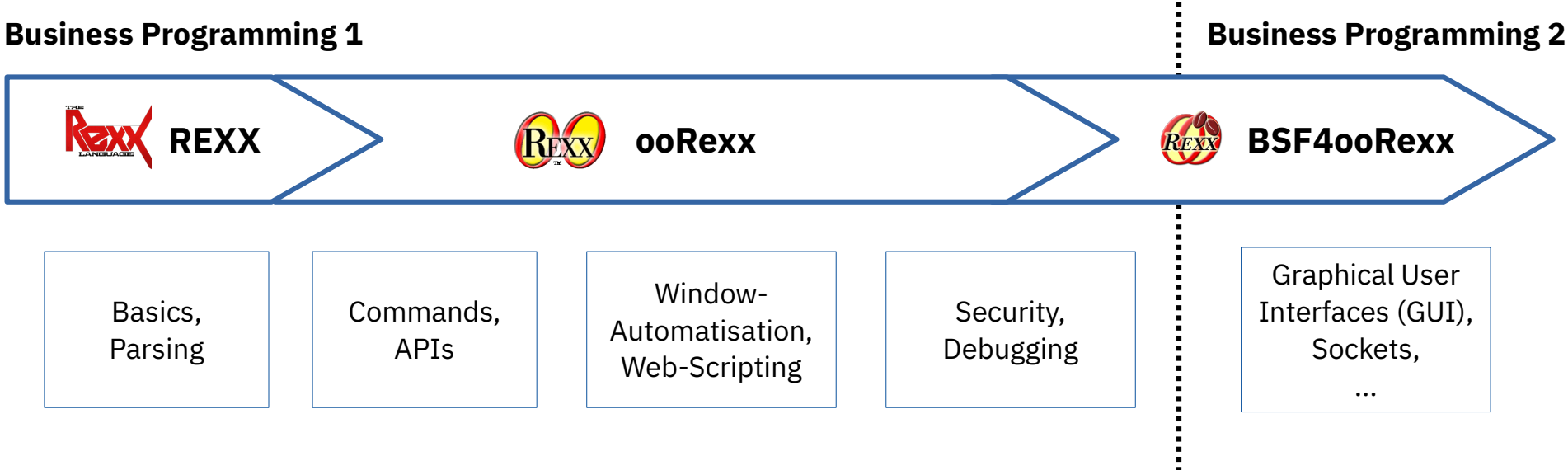
Procedural and Object-oriented Programming

Debugging, Multithreading, Security Manager, Off the Records



Prof. Rony G. Flatscher

Changelog TW, 2022-04-01 (Satz 6)

- Changelog:
- p. 2-6 Text Anpassungen
- p.10f → Commands
- p. 14f changes to Titel, text & animations
- Still needs extensions with cUrl example
- Multithreading, Security Manger, OffTopics moved to 07
- Kommentare:
- p 9. sort_util.cmd kommt mir falsch vor
- Class Methods vs. Instance Methods, 1 vernünftig darzustellen finde ich sehr schwer!

Changelog TW, 2022-04-01 (Satz 7)

- Comment:
  - Security Manager Example p.26f doesn’t exists any more

Multithreading, 1

- Multithreading
  - Parallel execution of different parts of an Object Rexx program
    - Parallel execution of methods
      - Multihreading **between** different objects: "**inter**-multithreading"
      - Multithreading **within** one and the same object: "**intra**-multithreading"
- Possible Problems
  - Accessing shared resources concurrently, e.g.
    - Concurrent alteration of attributes,
    - Concurrent alteration of files etc.
  - "Deadlocks", e.g.
    - Object 1 reserves: resource **A** and then **B**
    - Object 2 reserves: resource **B** and then **A**

Prof. Rony G. Flatscher

Multithreading, 1

- Multithreading
  - Parallel execution of different parts of an Object Rexx program
    - Parallel execution of methods
      - Multihreading **between** different objects: "**inter**-multithreading"
      - Multithreading **within** one and the same object: "**intra**-multithreading"
- Possible Problems
  - Accessing shared resources concurrently, e.g.
    - Concurrent alteration of attributes,
    - Concurrent alteration of files etc.
  - "Deadlocks", e.g.
    - Object 1 reserves: resource **A** and then **B**
    - Object 2 reserves: resource **B** and then **A**

Multithreading, 2

- Object Rexx default behaviour (continued)
  - All methods are **GUARD**ed by default, hence access to attributes is serialized
    - Within a class by default only one method can be executed for one and the same object, as that method gets **exclusive** access to the attributes, **blocking** all other methods of that class
    - Methods of one and the same object defined in different classes, are able to run concurrently (intra-multithreading) as each of these methods accesses attributes at their class level
- The keyword **UNGUARD** of a method directives allows that method to run concurrently with any other method in that class for one and the same object
  - There is no exclusive access protection to the objects!
  - May make sense, if attributes are not accessed at all or are not changed

Cf. rexxref.pdf (12.4. Using Additional Concurrency Mechanisms)

Multithreading, 3

- Object Rexx default behaviour (continued)
  - It is possible to determine at runtime whether methods are executed concurrently with other methods of the same class for one and the same object
    - REPLY statement of a method
      - Same effect as the RETURN statement
        - Calling program receives execution control (continues to run), **but**
        - **In addition** the method continues to run (concurrently)!
    - Optionally the REPLY statement may return a value to the calling program
      - If a REPLY statement has a return value, then in that method a RETURN statement must not supply a return value later on

Cf. rexxref.pdf (2.24. REPLY, 2.25. RETURN)

Multithreading, 4

- It is possible to determine at runtime whether methods are allowed to be executed concurrently with other methods of the same class for one and the same object
  - GUARD
    - GUARD ON statement
      - **Exclusive access to the attributes is desired**; if another method has already exclusive access, then execution is halted until the other method releases it
    - The GUARD OFF statement releases the exclusive access to the attributes
  - Efficient safeguarding of "critical segments"
    - Waiting for exclusive access can be made dependent on a given value appearing in the attributes of the object
    - Waiting for releasing the exclusive access can be made dependent on a given value appearing in the attributes of the object

Cf. rexxref.pdf (12.4. Using Additional Concurrency Mechanisms)

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
```

Output:

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

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
```

Output:

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

```
::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
```

Class MESSAGE

- **Message** class
  - Two possibilities to dispatch messages
    - SEND - synchronous execution
      - Execution proceeds, after the message was completely carried out
    - START - asynchronous execution (multithreading)
      - Message is dispatched and causes the activation of the method
      - Execution of the calling program proceeds concurrently
  - Additional interesting methods in the Message class
    - COMPLETED - indicates, whether an asynchronously excecuting method has completed
    - RESULT - waits for and returns the result of an asynchronously executing method
    - NOTIFY - allows sending a message to an object to notify it that a message has finished executing

Cf. rexxref.pdf (5.1.2. Message Class)

Using Class MESSAGE, no 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
```

Output:

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


Cf. rexxref.pdf (5.1.2. Message Class)

Using OBJECT's START-method, no REPLY!

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

Output:

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


Cf. rexxref.pdf (5.1.2. Message Class)

Multithreading, 10

- Executing threads concurrently
  - How to determine whether all concurrently executing threads have stopped?
- Example class **Waiter**
  - Simple class whose only instance method "wait" is to run in the background for a random length of time
  - Number of running threads is counted with a class attribute
  - Class method "wait" blocks until counter drops to 0 and returns then to the caller/invoker
  - Original idea and code: cf. Ian Collier, news:comp.lang.rexx, 2004-11-09

Class WAITER, Waiting on Threads ...

```
w=.waiter~new -- create an instance
do i=1 to 5
   w~wait(i)  -- invoke instance method
end
say "Waiting for counter to drop to 0..."
.waiter~wait  -- invoke class method
say "All done"

/* Waiter */
::class waiter
::method init class  -- class method
   expose counter
   counter=0         -- set initial value
::method up   class  -- class method
   expose counter
   counter=counter+1 -- increase counter
::method down class  -- class method
   expose counter
   counter=counter-1 -- decrease counter
::method wait class  -- class method
   expose counter
   -- wait until counter drops to 0
   guard on when counter=0
::method wait unguarded -- instance method
   a=random(1,6)  -- get a number between 1 and 6
   reply          -- now concurrency starts
   parse arg n    -- get invocation number
   .waiter~up     -- increase counter
   if n\='' then say 'Waiter' n 'waiting' a 'seconds'
   call syssleep a   -- sleep a few seconds
   if n\='' then say 'Waiter' n 'finished'
   .waiter~down   -- decrease counter
```

Possible Output:

```
Waiting for counter to drop to 0...
All done
Waiter 5 waiting 4 seconds
Waiter 1 waiting 2 seconds
Waiter 3 waiting 5 seconds
Waiter 4 waiting 4 seconds
Waiter 2 waiting 1 seconds
Waiter 2 finished
Waiter 1 finished
Waiter 5 finished
Waiter 4 finished
Waiter 3 finished
```

Security Manager, 1

- Allows supervising and intercepting
  - Access to the **environment**
    - ADDRESS statement
    - Messages to .local or .environment
  - Invoking **external** programs, routines (procedures/functions)
    - Invoking **external** programs or public routines with the CALL statement or with the help of the ::REQUIRES directive
  - Sending protected messages
    - Keyword PROTECTED in the method directive, or
    - Dynamically at runtime: sending the SETPROTECTED message to the method object
  - Interaction with Stream objects, e.g. using the messages
    - CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, LINES, STREAM

Cf. rexxref.pdf (13. The Security Manager)

Security Manager, 2

- This allows to monitor (Object) Rexx programs and inhibit - direct or indirect - access of the environment and external resources!
  - Intercepted (function) calls or messages can
    - be **replaced** by **self defined** routines or message routines
      - "Invisible": the supervised program is not able to note that another routine (function/procedure) got invoked or another message was sent in place of the original one!
    - lead to a **controlled** access violation which **tears down** the supervised program
    - be allowed to execute

Cf. rexxref.pdf (13. The Security Manager)

Security Manager, 3

- Enables rather interesting applications
  - Creation of execution profiles for (Object) Rexx programs
  - Creation of company related and task centric supervised execution environments (e.g. "sandbox") for (Object) Rexx programs
    - Secured execution of (Object) Rexx programs, which stem from anonymous or unsecure sources, e.g.
      - "Roaming Agents", which are transmitted/distributed via the Internet
  - Logging of routine (function/procedure) and/or message routine invocations, which are regarded to be important
  - ...

Cf. rexxref.pdf (13. The Security Manager)

Security Manager, 4

- Course of action
  - The program wishing to employ the Security Manager creates a method object by sending the NewFile message to the class **METHOD**, supplying the name of the program to be supervised
  - The method object gets assigned a supervisor object by sending it the SetSecurityManager message, supplying the supervisor object
  - After activating the method object, the runtime system's Security Manager sends the supervisor object the following messages
    - CALL, COMMAND, REQUIRES, LOCAL, ENVIRONMENT, STREAM, METHOD
      - Every Security Manager message receives as an argument a directory object, which contains supplemental information and which may be used to communicate with the Security Manager
      - Each method **must** return either 1 (supervisor program carried out the desired action already) or 0 (carry out the desired action)

Cf. rexxref.pdf (13. The Security Manager)

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
```


Cf. rexxref.pdf (13.1.1. Example)

Supervisor-Programm: "No Way!"

- For the supervised program a method object is created using the NewFile message of the class **METHOD**, which then gets an instance of the supervisor class "**noWay**" assigned to
- In this example the method object containing the supervised program will be activated (run) with the help of an agent object, which gets the supervised method object assigned to under the name "DISPATCH", hence sending that message to this one-off object runs the method

```
/* 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 object 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!")
```


Cf. rexxref.pdf (13.1.1. Example)

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         */
```


Cf. rexxref.pdf (13.1.1. Example)

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     */
```


Cf. rexxref.pdf (13.1.1. Example)

UNKNOWN Method

- If a method cannot be found,
  - then the UNKNOWN method is invoked, if defined in one of the searched classes
    - The runtime system supplies two arguments
      - Name of the message for which no method could be found
      - An array object containing the supplied arguments to the message

```
/* A possible UNKNOWN method */
::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
```

  - otherwise the runtime system raises the NOMETHOD exception
    - If no exception handling is defined for this, the program will be aborted with the message "object cannot understand message"

Cf. rexxref.pdf (4.2.6. Defining an UNKNOWN Method)

FORWARD Statement

- "Redirection of messages"
  - Changing the target object of the message (TO)
  - Changing the starting class for searching for the method using one of the available superclass objects (CLASS)
  - Changing of the message name (MESSAGE)
  - Forwards the accompanying arguments unchanged, except if
    - ARGUMENT or
    - ARRAY is given
  - Returns afterwards to the original sender of the message, unless
    - CONTINUE is given

Cf. rexxref.pdf (2.8. FORWARD)

Metaclasses

- Metaclasses
  - Object Rexx class **Class** and all of its subclasses, if any
  - Allow maintaining the methods, instances of such a class should get assigned to (Methods DEFINE, DELETE, METHOD, METHODS)
  - Allows creating objects (instances) from a class (Method NEW)
  - Instance of a metaclass is called "class object"
  - It is always possible to get access to the class object which created an instance/object by sending it the message "CLASS"
    - Method CLASS defined in the root class "Object" will get invoked and will return the appropriate class object
- "Class attributes" – Attributes of a metaclass
- "Class methods" – Methods of a metaclass

Cf. rexxref.pdf (4.1.4. Metaclasses)

Class Methods, 1

- Class methods
  - Methods of a metaclass
    - All methods of the Object Rexx class **Class**, e.g.
      - ID, DEFINE, DELETE, METHODS
    - All methods of the Object Rexx root class **Object**, e.g.
      - STRING, HASMETHOD
    - All methods, which are defined with a ::METHOD directive containing the keyword CLASS are assigned to the class object
      - The interpreter uses SETMETHOD from the Object Rexx root class **Object**
  - Class methods can be invoked via the class object
    - Each object can retrieve its own class object by sending itself the message Class and thereafter sending the desired messages to the class object

Class Methods, 2

```
/**/
SAY COPIES("-", 50)
.Test~Hallo_1
SAY COPIES("-", 50)
o = .Test~New
o~Hallo_2
::CLASS  Test
::METHOD Init  CLASS
  SAY "New class [" || self~string || "] is being created..."
  self~init:super
::METHOD Hallo_1 CLASS
  SAY "Hallo, I am [" || self~string || "]..."
::METHOD Init
  SAY "New instance ["self~string"] is being created..."
  self~init:super
::METHOD Hallo_2
  SAY "Hallo, I am ["self~string"]..."
```

Output:

```
New class [The TEST class] is being created...
------------------------------------------------
Hallo, I am [The TEST class]...
------------------------------------------------
New instance [a TEST] is being created...
Hallo, I am [a TEST]...
```

Class Attribute, 1

- Attributes of metaclasses
  - Access like any other attributes
    - Attribute methods
      - Method directive with the key word ATTRIBUTE
      - Allow getting and setting the value
      - Using the EXPOSE statement as the first statement in a method

Class Attribute, 2

```
/**/
.Test ~~New ~~New ~~New ~~New ~~New ~~New
SAY "So far, there have been [".Test~Counter"] objects created."
o = .Test~New
SAY "So far, there have been ["o~class~Counter"] objects created."
SAY "class:" o~class~string", last instance was:" o~string

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

Output:

```
So far, there have been [6] objects created.
So far, there have been [7] objects created.
class: The TEST class, last instance was: a TEST
```

Metaclass, 1

- If the runtime system (interpreter) finds a ::CLASS directive, it creates an instance of type **Class** ("class object") for it
- If the runtime system (interpreter) finds a ::METHOD directive, it creates an instance of type **Method** for it
  - Method objects which are defined for instances of a class are called "**instance** methods" and are stored in the class object with the help of its DEFINE method
    - One can retrieve a supplier of all defined instance methods by sending the METHODS message to the class object
  - Method objects, which are meant for the class object itself (method directive containing the keyword CLASS) are dubbed "**class** methods" and are attached to the class object with the help of SETMETHOD which is defined in the Object Rexx root class **Object**

Cf. rexxref.pdf (4.1.4. Metaclasses)

Metaclass, 2

- Object Rexx programs can be devised, which create class objects and method objects at runtime ("dynamic")
  - Classes are represented as class objects (instances of the Object Rexx class **Class**)
  - Methods are represented as method objects (instances of the Object Rexx class **Method**)
    - Instance methods are stored with the class object using its DEFINE method (one could remove an instance method from it with DELETE)
      - One can retrieve a supplier of all defined instance methods by sending the METHODS message to the class object
    - Class methods can be directly assigned to class objects using SETMETHOD- defined in the Object Rexx root class **Object** (one could remove a class method with UNSETMETHOD)

Cf. rexxref.pdf (4.1.4. Metaclasses)

Metaclass, 3

- The metaclass **Class** normal Object Rexx class
  - Therefore it can be subclassed (specialized)
  - All subclasses of **Class** are metaclasses themselves !
  - Should a specialized metaclass be used for creating the class object, then the ::CLASS directive must contain the keyword METACLASS followed by the name of the desired metaclass
    - The default is: METACLASS Class
- Sending a message by the name Class (available as method in the root class **Object**) to an object will always return its class object (instance of a metaclass)
  - Hence all public methods of the metaclass are always available to an object via its class object

Cf. rexxref.pdf (4.1.4. Metaclasses)

Creating the Class Object ".TEST"

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

::CLASS  Test
::METHOD Init  CLASS
  SAY "New class [" || self~string || "] is being created..."
  self~init:super
::METHOD Hallo_1 CLASS
  SAY "Hallo, I am [" || self~string || "]..."
::METHOD Init
  SAY "New instance ["self~string"] is being created..."
  self~init:super
::METHOD Hallo_2
  SAY "Hallo, I am ["self~string"]..."
```

Output:

```
New class [The TEST class] is being created...
------------------------------------------------
Hallo, I am [The TEST class]...
------------------------------------------------
New instance [a TEST] is being created...
Hallo, I am [a TEST]...
```

```
.TEST [defined as: .class~new("TEST")]
```

```
BASECLASS
DEFAULTNAME
DEFINE
DELETE
ENHANCED
ID
INHERIT
INIT
METACLASS
METHOD
METHODS
MIXINCLASS
NEW
QUERYMIXINCLASS
SUBCLASS
SUBCLASSES
SUPERCLASSES
UNINHERIT
INIT
HALLO_1
```

```
o = .Test~New
INIT, HALLO_2

o1 = .Test~New
INIT, HALLO_2

o2 = .Test~New
INIT, HALLO_2

o3 = .Test~New
INIT, HALLO_2
```


Cf. rexxref.pdf (4.2. Creating and Using Classes and Methods)

Creating the Class Object ".TEST"

Directives are encountered by the runtime system (interpreter)

Directives are carried out by the runtime system (interpreter)

```
::CLASS  Test                  -- class

::METHOD Init  CLASS           -- method #1
  SAY "New class [" || self~string || "] is being created..."
  self~init:super


::METHOD Hallo_1 CLASS         -- method #2
  SAY "Hallo, I am [" || self~string || "]…"


::METHOD Init          -- method #3
  SAY "New instance ["self~string"] is being created..."
  self~init:super


::METHOD Hallo_2               -- method #4
  SAY "Hallo, I am ["self~string"]...“
```

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

meth1=.method~new("INIT", source1)     -- create method #1
.test~setmethod("INIT", meth1)         -- add to class object


meth2=.method~new("HALLO_1", source2) -- create method #2
.test~setmethod("HALLO_1", meth2)      -- add to class object


meth3=.method~new("INIT", source3)     -- create method #3
.test~define("INIT", meth3)            -- define as instance method


meth4=.method~new("HALLO_2", source4) -- create method #4
.test~define("HALLO_2", meth4)         -- define as instance method
```


Cf. rexxref.pdf (4.2. Creating and Using Classes and Methods)

Creating the Class Object ".TEST"

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

::CLASS  Test
::METHOD Init  CLASS
  SAY "New class [" || self~string || "] is being created..."
  self~init:super
::METHOD Hallo_1 CLASS
  SAY "Hallo, I am [" || self~string || "]..."
::METHOD Init
  SAY "New instance ["self~string"] is being created..."
  self~init:super
::METHOD Hallo_2
  SAY "Hallo, I am ["self~string"]..."
```

Output:

```
New class [The TEST class] is being created...
------------------------------------------------
Hallo, I am [The TEST class]...
------------------------------------------------
New instance [a TEST] is being created...
Hallo, I am [a TEST]...
```

```
.TEST  [.source~test= .class~new("TEST")]
```

```
BASECLASS
DEFAULTNAME
DEFINE
DELETE
ENHANCED
ID
INHERIT
INIT
METACLASS
METHOD
METHODS
MIXINCLASS
NEW
QUERYMIXINCLASS
SUBCLASS
SUBCLASSES
SUPERCLASSES
UNINHERIT
INIT
HALLO_1
```

```
o = .Test~New
INIT, HALLO_2

o1 = .Test~New
INIT, HALLO_2

o2 = .Test~New
INIT, HALLO_2

o3 = .Test~New
INIT, HALLO_2
```

Defining Metaclasses, 1

- Some problems can be elegantly solved with the help of metaclasses
  - Example: **Singleton**
    - Ensure that there is only one instance created from a class !
- Creating objects is realized via the class object's NEW method
  - Hence, if it was possible to check in the appropriate NEW method whether it already created and returned an instance, then
    - One can inhibit the creation of additional instances by inhibiting the forwarding of the NEW message to the superclass, **and**
    - It becomes possible to return the already created (single) instance instead, if it got stored in a class attribute.
- For this purpose a metaclass Singleton shall be defined
  - If one would need a class with this singleton behaviour, then it would be sufficient for it to merely extend the ::CLASS directive with the metaclass keyword and indicate the desired metaclass: **METACLASS Singleton**

Defining Metaclasses, 2

- Object Rexx implementation of the metaclass 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
```

Defining Metaclasses, 3

- There is an abstract datatype defined for an Easter bunny
  - Attribute: **usageSite**
  - Methods: getting/setting values for the attribute usageSite
- As there is can be only one Easter bunny it must be made sure, that only one instance can be created !
  - Therefore the class object for **EasterBunny** should be created from the metaclass **Singleton**, which makes sure that this behaviour is enforced

Defining Metaclasses, 4

```
/* EasterBunny */
a = .EasterBunny~new("Vienna, Austria")
b = .EasterBunny~new("Stumm im Zillertal")
SAY "a==b:" (a==b) "usageSite of b:" b~usageSite
::CLASS  EasterBunny  METACLASS Singleton
::METHOD usageSite ATTRIBUTE
::METHOD Init
  self~usageSite = ARG(1)
  SAY "Init-method: usageSite is:" self~usageSite
  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
```

Output:

```
Init-method: usageSite is: Vienna, Austria
a==b: 1 usageSite of b: Vienna, Austria
```

Defining Metaclasses, 5

- In Example 2 there was a test class which counted the number of created objects
- Define a metaclass **Counter**, which
  - Counts how many objects have been created, and
  - Which can be interrogated for the total of the created instances

```
::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)
```

Defining Metaclasses, 6

```
/**/
.Test ~~New ~~New ~~New ~~New ~~New ~~New
SAY "So far, there have been [".Test~Counter"] objects created."
o = .Test~New
SAY "So far, there have been ["o~class~Counter"] objects created."
SAY "class:" o~class~string", last instance was:" 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)
```

Output:

```
So far, there have been [6] objects created.
So far, there have been [7] objects created.
class: The TEST class, last instance was: a TEST
```

"ENHANCED" Method of "CLASS", 1

- Sometimes there is a need to create an instance of a class, which possesses all attributes and methods laid out in the class, but differs slightly in a few methods
  - Problem solution
    - One needs to define a subclass implementing the different methods
    - If there is a need to create many different objects, each differing slightly in a few methods, then this approach may become a little cumbersome
- The ENHANCED method allows the creation of an instance from an existing class which receives those methods which need to behave differently
  - Such objects are called "**one-off objects**"

Cf. rexxref.pdf (4.2.3. Defining Instance Methods...)

"ENHANCED" Method of "CLASS", 2

- The class PERSON has an attribute **Name** and the methods **your_name** and **from_where** which are language dependent, in addition to numerous other methods
  - The methods **your_name** and **from_where** shall be implemented in the national language of the individual persons
    - Default is the German language and the default country is Austria (Europe)
      - "Ich heiße ..."
      - "Ich komme aus Österreich."
    - English persons
      - "My name is ..."
      - "I am from ..."
    - Spanish persons
      - "Mi nom es ..."
      - "Soy de ..."

Cf. rexxref.pdf (4.2.3. Defining Instance Methods...)

"ENHANCED" Method of "CLASS", 3

```
/**/
es = .directory~new
es ~your_name = "RETURN 'Mi nom es' self~Name'.'"
es ~from_where= "RETURN 'Soy de España.' "
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(es, "Juan")
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."
```

Output:

```
Ich heiße Hans. Ich komme aus Österreich.
Mi nom es Juan. Soy de España.
My name is John. I am from America.
```

Initializing Object Rexx

- Interpreter is started
- The Object Rexx environments **.environment** and **.local** are created
- The Rexx program given as argument
  - Is checked for syntax errors
  - Directives are carried out, the **source** directory is created
    - **::REQUIRES** loads the specified program, its syntax is checked and its directives are carried out (a corresponding source directory is created), required program is executed starting at the first line
    - Class objects are created for the classes and are stored in the source directory
  - Execution of the program starts at the first line