An Introduction to Procedural and Object-oriented Programming (Object Rexx) 5

Defining Classes ("CLASS" Directive),
Defining Methods ("METHOD" Directive),
Object Rexx Classes, Object Rexx Collection Classes

Prof. Rony G. Flatscher

Abstract Datatype (ADT) Implementation with Object Rexx

- Abstract Datatype
  - **Schema** for the implementation of datatypes
    - **::CLASS** directive
      - Definition of **attributes** and therefore the internal data structure
        - **EXPOSE** statement **within** methods or
        - **::**METHOD directive with the keyword **ATTRIBUTE**
      - Definition of **operations** (functions, procedures)
        - ::METHOD directive
  - Instances of a class ("objects")
    - Individual, unambiguously distinguishable instantiations of the same type
    - Possesses all the same attributes (constitutes the data structure as defined in the class) and operations ("methods of the class")

Object Rexx
Messages (~, ~~)

- **Interaction** (activating of functions/procedures) **with objects** (instances) **exclusively** via messages, which are sent to objects
  - Names of messages are the names of the methods, which 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)
Implementing an ADT in Object Rexx (Example)

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

```
/**/
g1 = .Birthday~New
g1~Date= "20120901"
g1~Time= "16:00"
g2=.Birthday~New~~"Date="("20160229")~~"Time="("19:19")
SAY g1~date g2~date g1~time g2~time
```

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

Output:

```
20120901 20160229 16:00 19:19
```

Execution of Cascading Messages, 1

- Executing cascading messages in the RVALUE part

```
g2=.Birthday~New  ~~"date="('20080229') ~~"time="('19:19')
```

Is carried out by the interpreter as follows:

```
_rvalue = .Birthday~New                  -- regular message
          _rvalue ~"date="('20080229')   -- cascading message
          _rvalue ~"time="('19:19')      -- cascading message
g2=_rvalue                               -- assigning to LVALUE
```

Execution of Cascading Messages, 2

- Executing cascading messages in the RVALUE part

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

Is carried out by the interpreter as follows:

```
_rvalue = '20080229' ~"+"(1)-- normal message, value: '20080230'
          _rvalue ~"*"(987) -- cascading message
_rvalue = _rvalue ~"/"(2)   -- normal message, value: '10040115'
          _rvalue ~"+"(6)   -- cascading message
_rvalue = _rvalue ~"+"(1)   -- normal message, value: '10040116'
x=_rvalue                   -- assign result to LVALUE
                            -- x has the value: '10040116'
```

Overview of Scopes

- Rexx and Object Rexx
  - Standard scope
    - Labels, variables
  - Procedure scope
    - Variables in 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)

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"***
- **Always invoke the INIT method of the superclass!**

Abstract Datatype "Person"
Implementation of Constructor in Object Rexx

```
/**/
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
  self~init:super -- invoke constructor of superclass
::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"***

Abstract Datatype "Person"
Implementation of Destructor in Object Rexx

```
/**/
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
  self~init:super -- invoke constructor of superclass
::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.
```

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 land-born and water-born animals, where there exists a class "amphibians", which inherits directly from the land-born and water-born 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, 3

- 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 "**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
  self~bark:super
::CLASS  "LittleDog" SUBCLASS dog
::METHOD Bark
  SAY self~Name":" "wuuf" "-" self
```

Output:

```
Sweety: Wuff Wuff - a DOG
Grobian: WUFFF! WUFFF!! WUFFF!!! - a BIGDOG
Grobian: Wuff Wuff - 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*

::CLASS Directive

- This directive causes the interpreter to create a class
  - **::CLASS** xyz
    - A class with the identifier XYZ is created
- Keywords allow to ask for/determine additional features
  - PUBLIC
    - Optional, class can be seen outside the program in which it is defined
  - SUBCLASS, MIXINCLASS
    - Optional, default value: SUBCLASS Object
  - METACLASS metaclass
    - Optional, default value: METACLASS Class
  - INHERIT
    - Optional, allows indicating those classes which are inherited in addition: **multiple inheritance**

::CLASS Directive
Example: ADT "Vehicle", 1

```
 /**/
.RoadVehicle  ~new("Truck") ~drive
.WaterVehicle ~new("Boat")  ~swim

::CLASS  Vehicle
::METHOD name    ATTRIBUTE
::METHOD INIT
  self~name = ARG(1)

::CLASS  RoadVehicle   SUBCLASS Vehicle
::METHOD drive
  SAY self~name": 'I drive now...'"

::CLASS  WaterVehicle     SUBCLASS Vehicle
::METHOD swim
  SAY self~name": 'I swim now...'"
```

Output:

```
Truck: 'I drive now...'
Boat: 'I swim now...'
```

::CLASS Directive
Example: ADT "Vehicle", 2

```
 /* Multiple Inheritance */
.RoadVehicle      ~new("Truck")   ~drive
.WaterVehicle     ~new("Boat")    ~swim
.AmphibianVehicle ~new("SwimCar") ~show_off
```

```
::CLASS  Vehicle
::METHOD name    ATTRIBUTE
::METHOD INIT
  self~name = ARG(1)
```

```
::CLASS  RoadVehicle   MIXINCLASS Vehicle
::METHOD drive
  SAY self~name": 'I drive now...'"
```

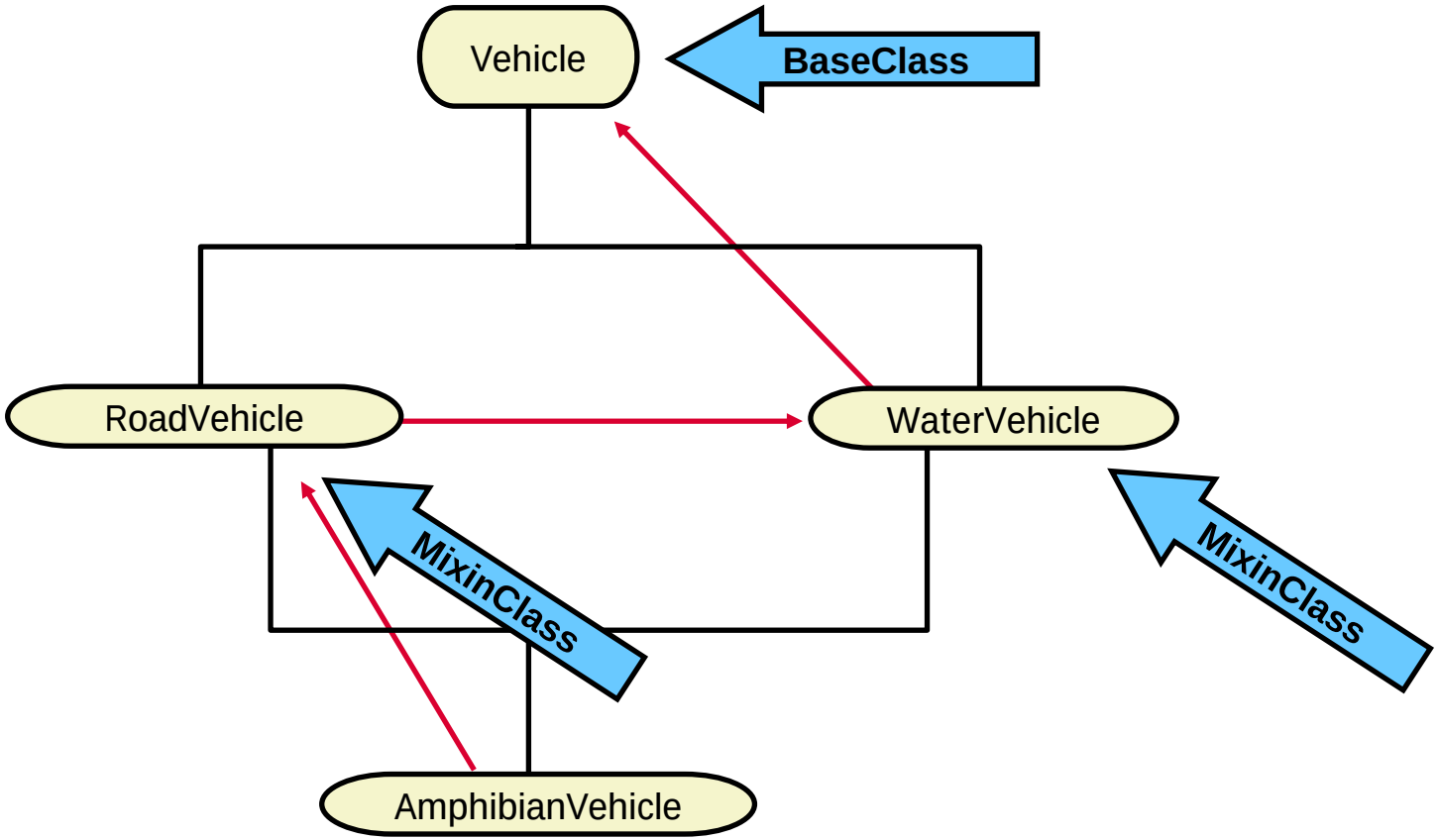
```
::CLASS  WaterVehicle     MIXINCLASS Vehicle
::METHOD swim
  SAY self~name": 'I swim now...'"
```

```
::CLASS AmphibianVehicle SUBCLASS RoadVehicle INHERIT WaterVehicle
::METHOD show_off
  self ~~drive ~~swim
```

Output:

```
Truck: 'I drive now...'
Boat: 'I swim now...'
SwimCar: 'I drive now...'
SwimCar: 'I swim now...'
```

::CLASS Directive
Example: ADT "Vehicle", 3



::METHOD Directive, 1

- This directive causes the interpreter to create a method
  - **::Method** mmm
    - A method with the identifier "MMM" is created
- Keywords allow to ask for/determine additional features
  - ATTRIBUTE
    - Optional, the interpreter creates **two** methods:
      - A get method "MMM" and
      - A set method "MMM=",
      - Which both access the **attribute MMM**

::METHOD Directive, 2

- ATTRIBUTE (continued)
  - The **get** method MMM" is defined as:

```
::METHOD MMM    /* name of get method "MMM"                    */
  EXPOSE MMM    /* allow direct access to the attribute        */
  RETURN MMM    /* return the attribute's value                */
```

  - The **set** "MMM=" is defined as:

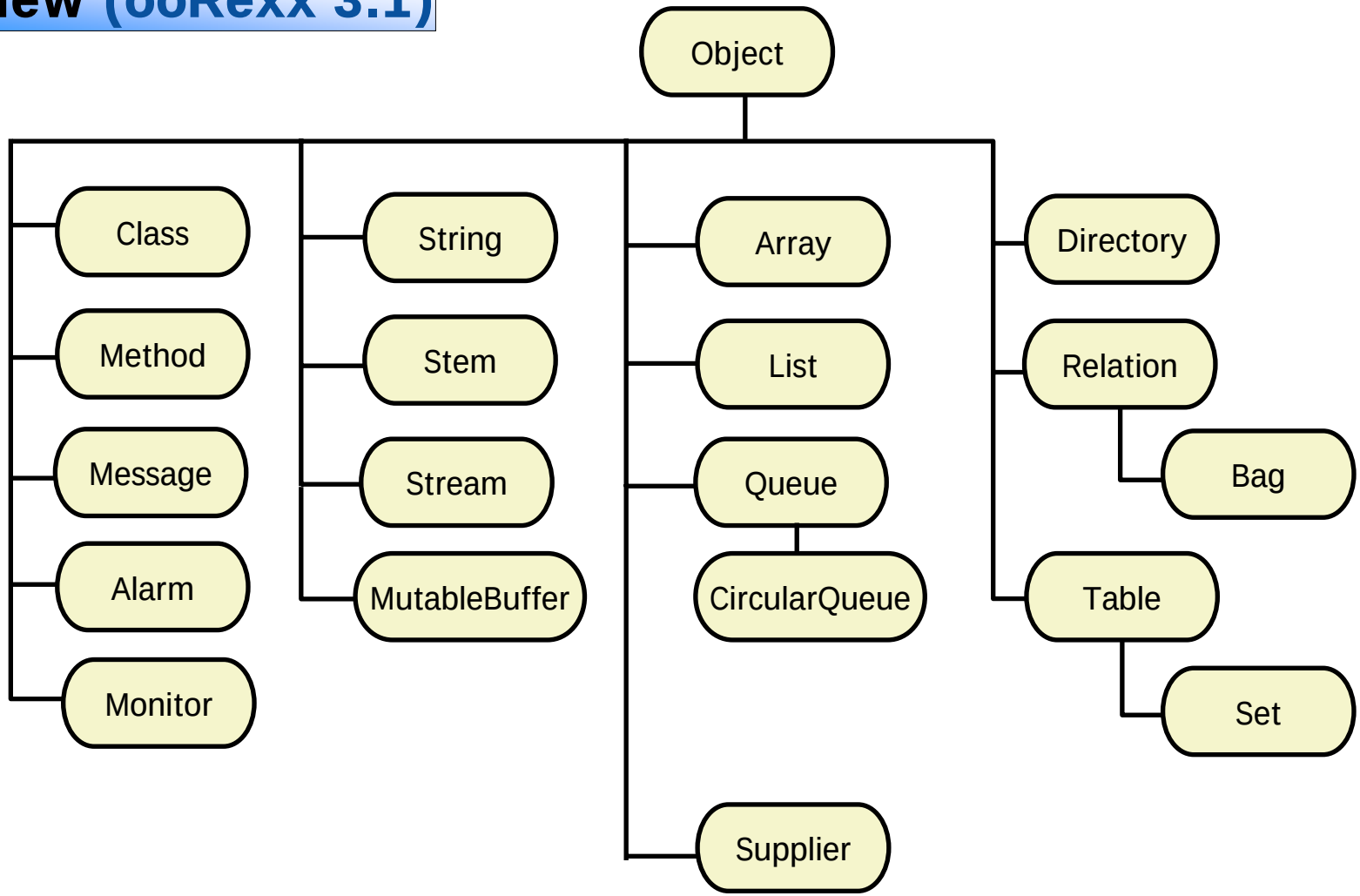
```
::METHOD "MMM=" /* name of the set method "MMM="               */
  EXPOSE  MMM   /* allow direct access to the attribute        */
  USE ARG MMM   /* retrieve argument and assign it to the
                   attribute                                   */
```

::METHOD Directive, 3

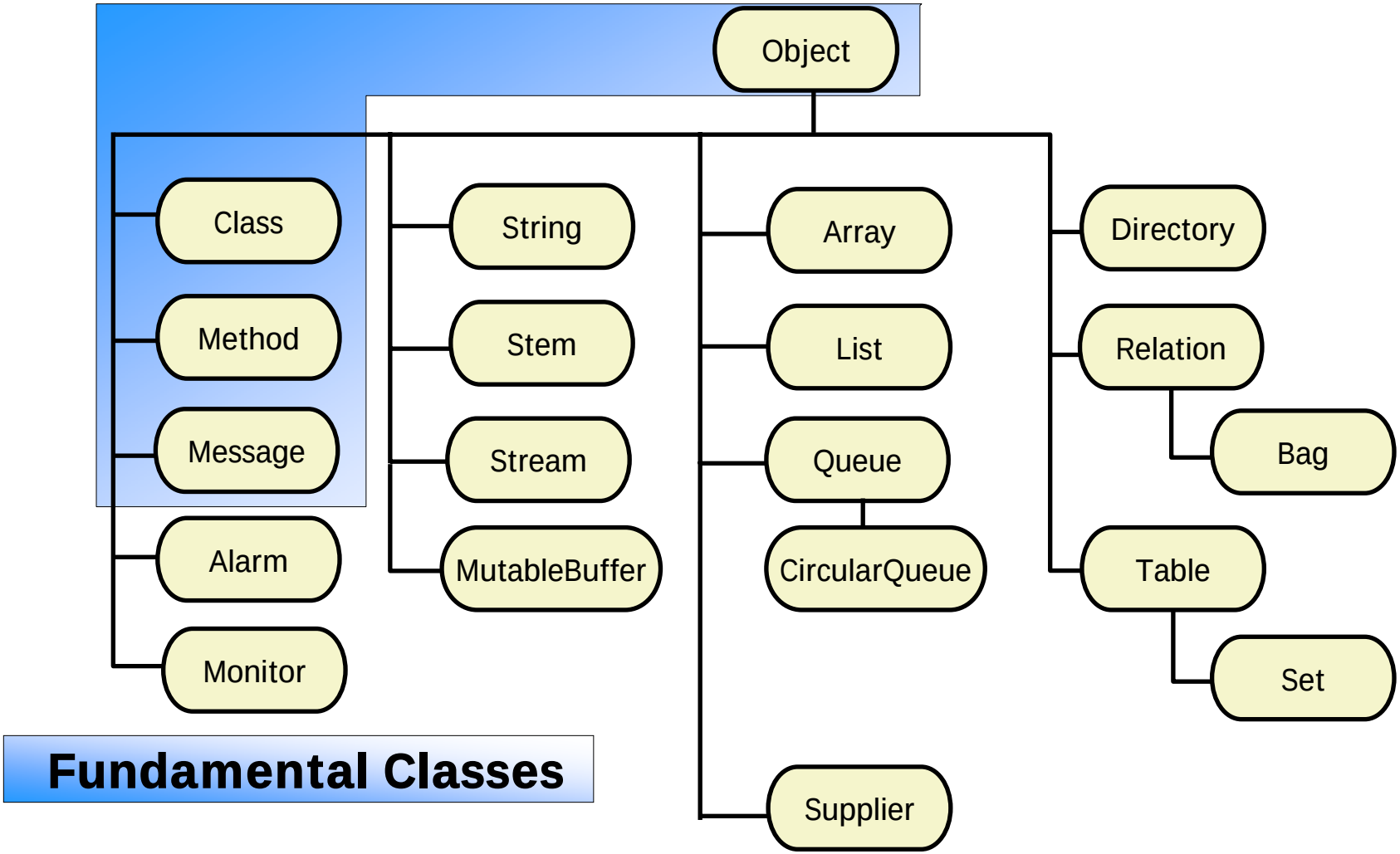
- Keywords allow to ask for/determine additional features
  - PRIVATE
    - Optional, such a method can only be activated from within an object:
      ```
      self~mmm
      ```
  - GUARDED, UNGUARDED
    - Optional, default value: GUARDED
    - Determines whether method can be run in parallel to other methods
  - CLASS
    - Optional, method is a class method
  - PROTECTED
    - Optional, access to this method can be supervised with the help of the Object Rexx Security Manager

Classification Tree of Object Rexx, 1

Overview (ooRexx 3.1)



Classification Tree of Object Rexx, 2



Fundamental Classes, 1

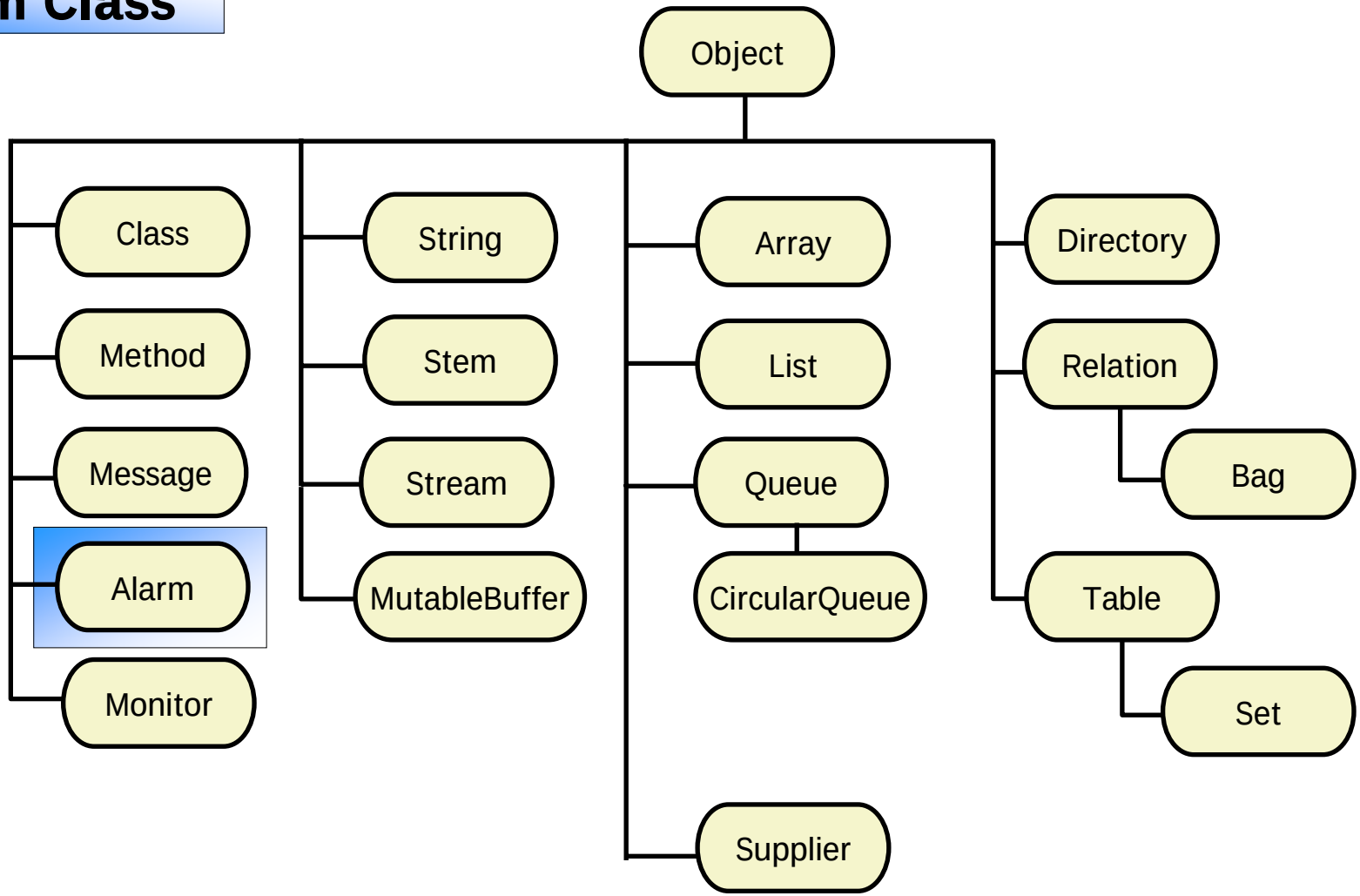
- Object
  - Methods and attributes are available to ***all*** Instances of Object Rexx Classes *(Objekte)*
    - Example: method INIT
      - Constructor, initializes a freshly created object
- Class
  - Interpreter creates an instance of this class ("class object") for each ::CLASS directive
    - Example: method ID
      - Returns the name (the "identification") of the class object
    - Example: method NEW
      - Returns a new instance (object) of the class

Fundamental Classes, 2

- Method
  - Interpreter creates an instance of this class ("method object") for each ::METHOD directive
    - Example: method SOURCE
      - Returns the source code of the method, if available
- Message
  - For each message at runtime the interpreter creates an instance of this class ("method object")
    - Example: method SEND
      - Transmits the message to the object

Classification Tree of Object Rexx, 3

Alarm Class

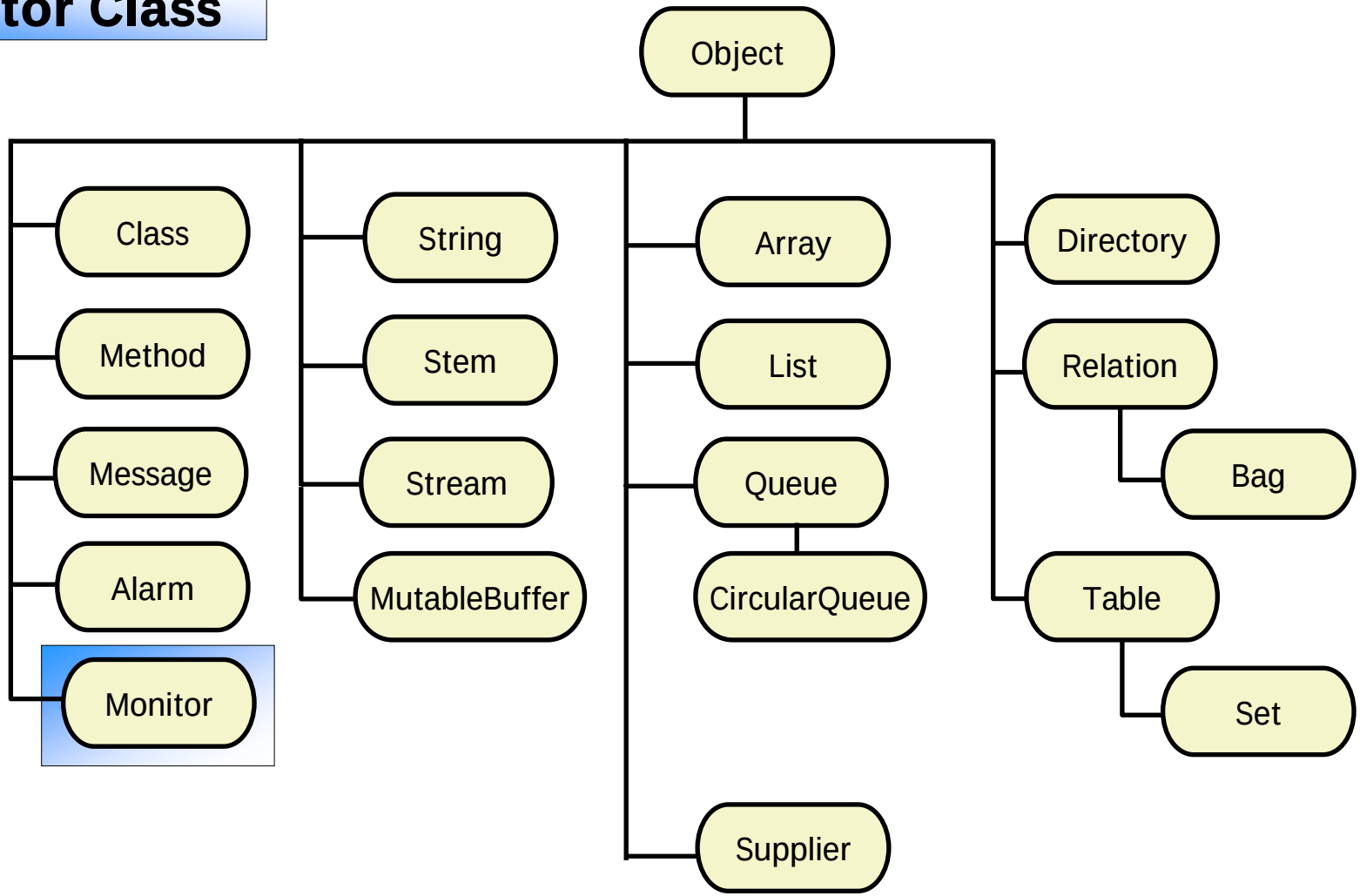


Alarm Class

- Alarm
  - Alarm objects allow dispatching messages at a later time
    - Such messages are carried out in parallel to other activities in the Object Rexx program ("multithreaded execution")
    - Dispatch time can be given
      - In hours, minutes, seconds starting from the time of initialization of the alarm object
      - As a time and date
    - Example: method CANCEL
      - Cancels an alarm object, the pending message will not be dispatched

Classification Tree of Object Rexx, 4

Monitor Class

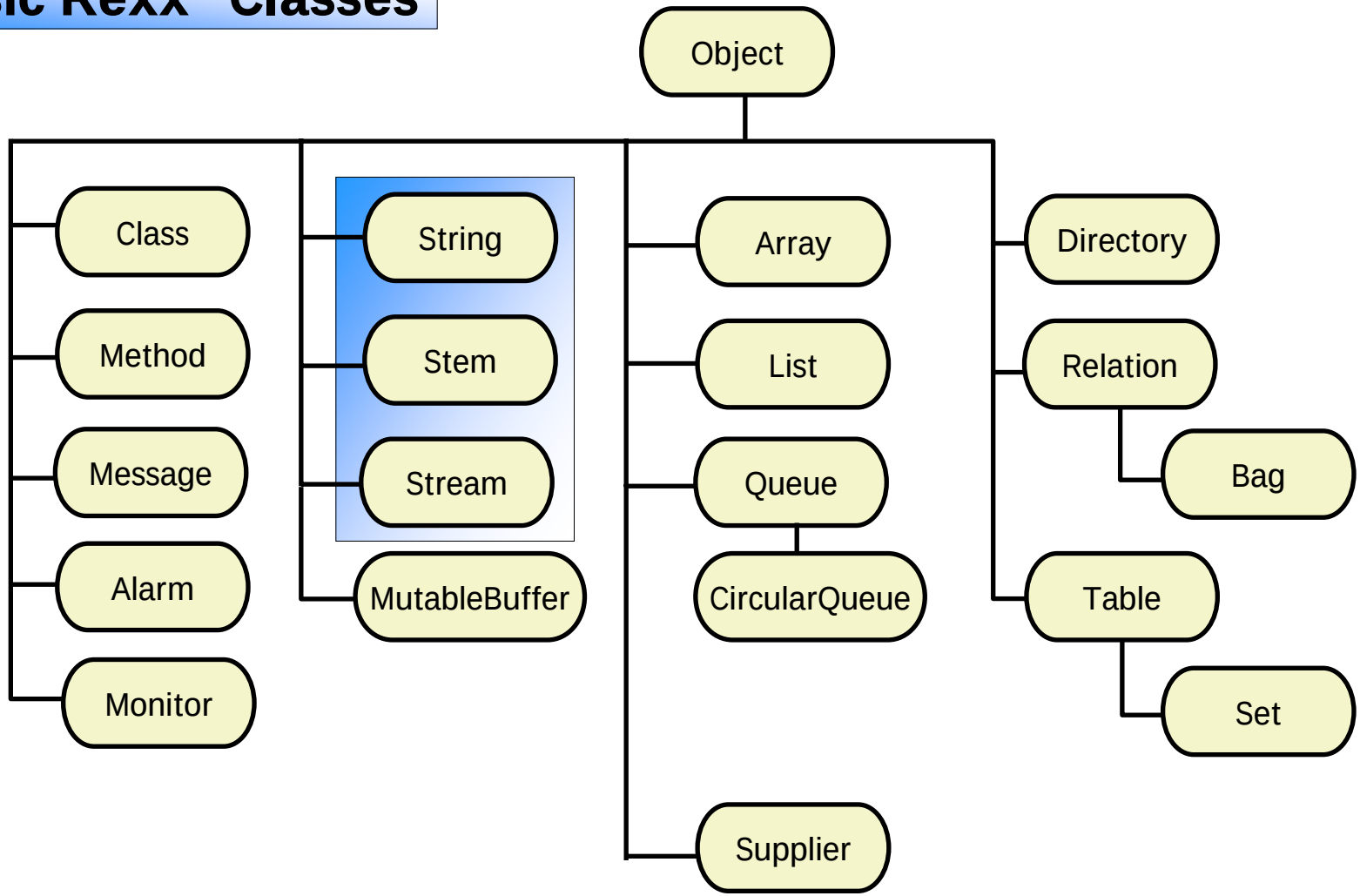


Monitor Class

- Monitor
  - Monitor objects allow the monitoring of messages sent to the object
    - Example: method DESTINATION
      - Determines the object to be monitored
      - Returns the object which gets monitored

Classification Tree of Object Rexx, 5

"Classic Rexx" Classes



"Classic Rexx" Classes, 1

- String (1)
  - String objects possess all methods, which are the counterparts of all string functions in classic Rexx
    - **Distinctive feature:** string objects *never* change the value they were created with!

```
a = .string~new("hallo") /* a new string object                    */
a = "hallo"      /* a new string object with a value of "hallo" */
a = "aloha"      /* a new string object with a value of "aloha" */
a = "aloha"      /* a new string object with a value of "aloha" */

a = "a" || "b"   /* a new string object with a value of "ab"    */
a = a   || "b"   /* a new string object with a value of "abb"   */

a = 1 + 3        /* a new string object with a value of "4"     */
a = a + 3        /* a new string object with a value of "7"     */
```

"Classic Rexx" Classes, 2

- String (2)
  - String functions will be transformed "behind the curtain" by Object Rexx into the appropriate object-oriented version, by sending the appropriate messages to the string object!
  - Example: method REVERSE
    - Reverses the sequence of characters in a string

```
SAY REVERSE("d:\path\datei.typ") /* function */
SAY "d:\path\datei.typ"~REVERSE  /* message  */
```

Output:

```
pyt.ietad\htap\:d
pyt.ietad\htap\:d
```

"Classic Rexx" Classes, 3

- Stem (1)
  - Stem objects allow any string to be used as an index
    - The stem of the identifier includes the first dot

```
a.2 = "I am a.2"
SAY a.1.b "/and\" a.2
```

Output:

```
A.1.B /and\ I am a.2
```

```
a.  = "no value"
a.2 = "I am a.2"
SAY a.1.b "/and\" a.2
```

Output:

```
no value /and\ I am a.2
```

```
a = .stem~new("no value") /* new stem object */
a[2] = "I am a.2"
SAY a[a.1.b] "/and\" a[2]
```

Output:

```
no valueA.1.B /and\ I am a.2
```

"Classic Rexx" Classes, 4

- Stem (2)
  - Stem objects allow the collection of arbitrary objects with the help of string indices
    - Example: methods [] and []=

```
DROP a a. b b. /* Make sure that variables are deleted */
a = .stem~new("xyz")
a["holladi"] = "Entry for 'holla.di'"
b. = a /* two references to the same stem object!  */
b.di.di.dumm = "Entry for 'DI.DI.DUMM'"
SAY "1:" a["holladi"]      "/and\" a~"[]"("DI.DI.DUMM")
tmp1 = "holladi"; tmp2 = "DI.DI.DUMM"
SAY "2:" a.tmp1            "/and\" a.[tmp2]
SAY "3:" b.tmp1            "/and\" b.[tmp2]
SAY "4:" a a. a.Unknown b b. b.Unknown a[Unknown]
```

Output:

```
1: Entry for 'holla.di' /and\ Entry for 'DI.DI.DUMM'
2: A.holladi /and\ A.DI.DI.DUMM
3: Entry for 'holla.di' /and\ Entry for 'DI.DI.DUMM'
4: xyz A. A.UNKNOWN B xyz xyzUNKNOWN xyzUNKNOWN
```

"Classic Rexx" Classes, 5

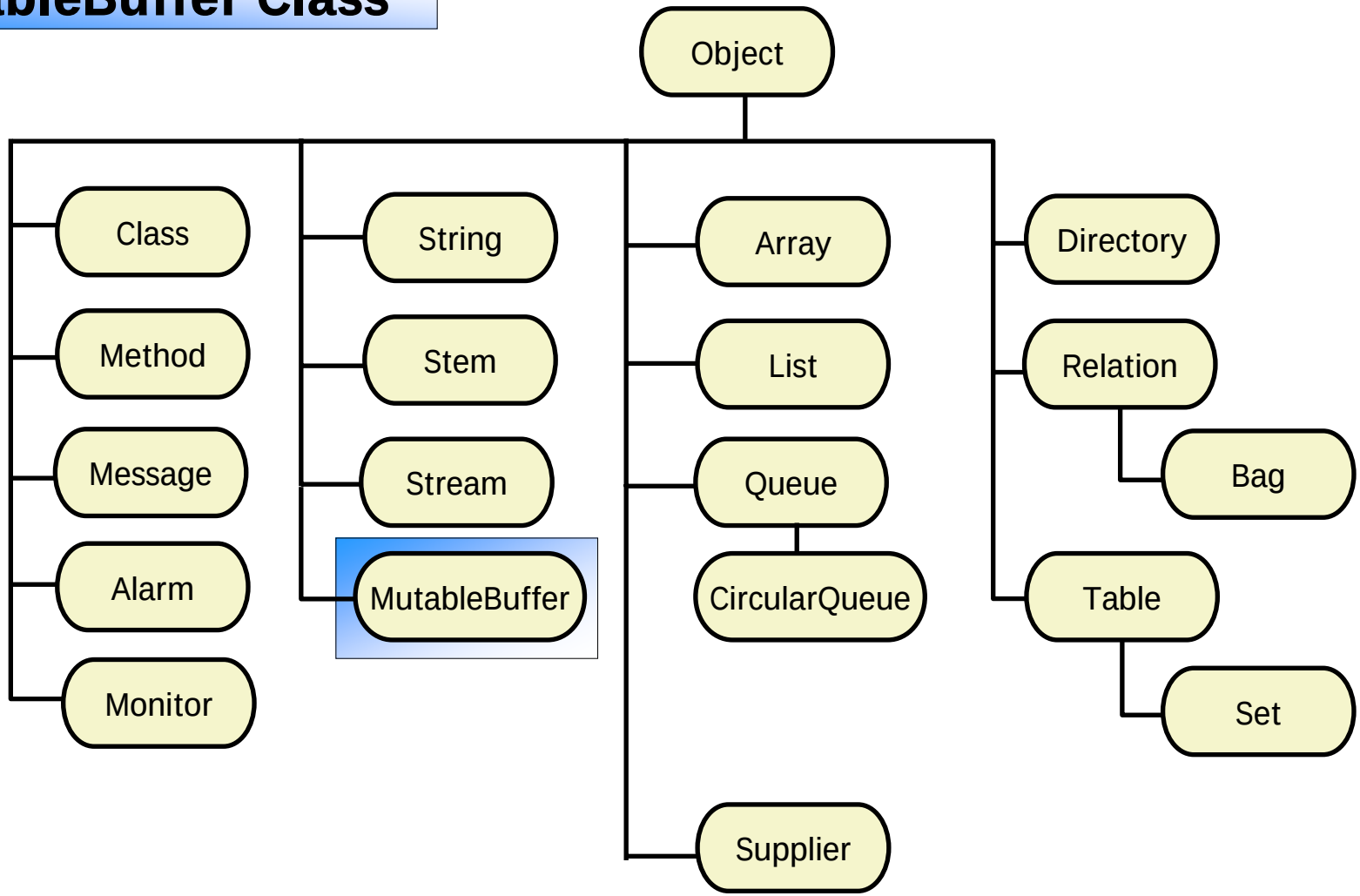
- Stream
  - Stream objects allow working with files
    - Example: method NEW

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

      - Allows working with the file **test.dat** by sending the stream object **o** the appropriate messages, e.g. OPEN for opening, LINEIN (CHARIN) for reading from the file, LINEOUT (CHAROUT) for writing to the file, CLOSE for closing

Classification Tree of Object Rexx, 6

MutableBuffer Class

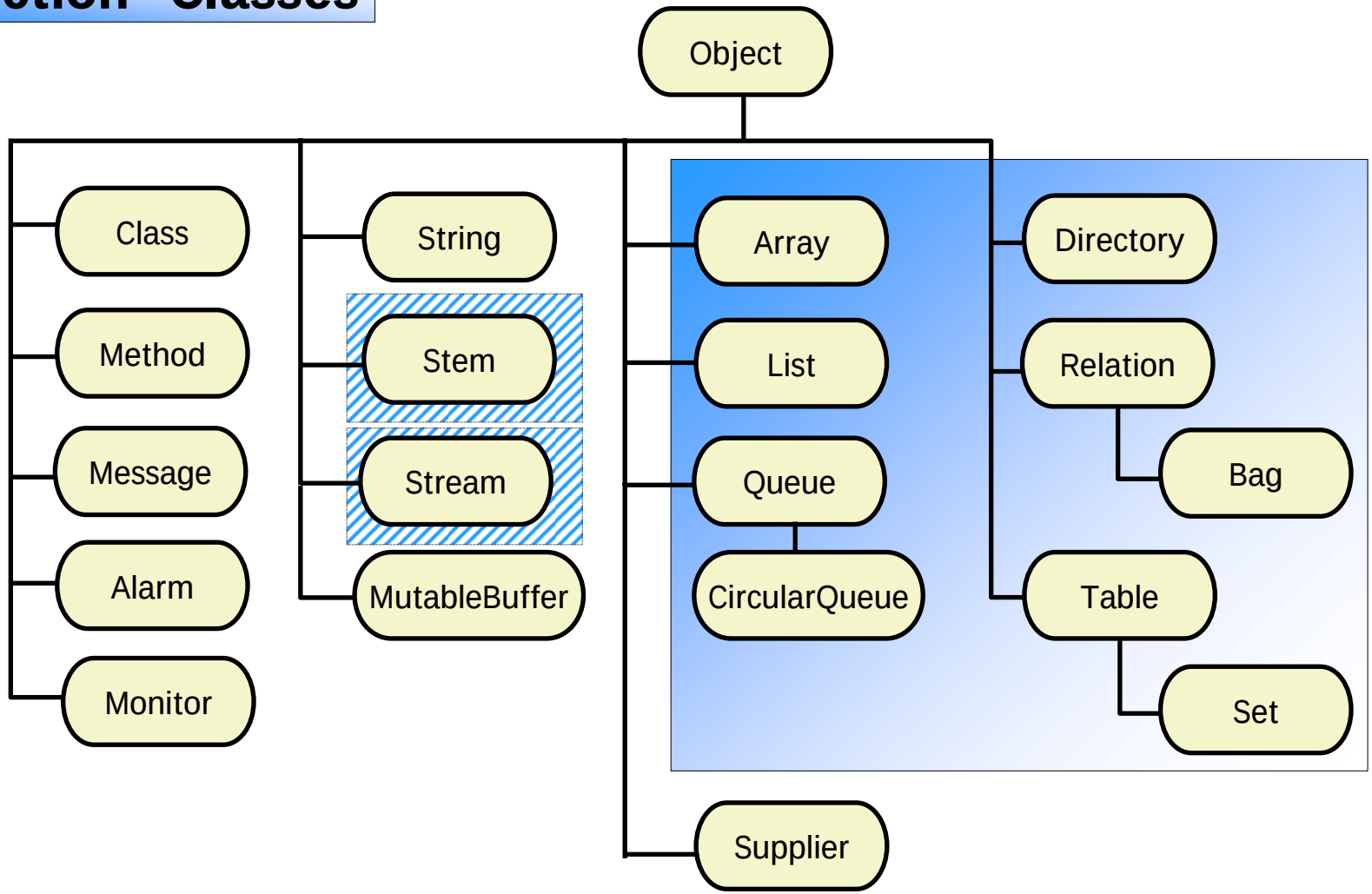


MutableBuffer Class

- MutableBuffer
  - Class that allows to create a buffer of strings quickly
    - Allows creating larges strings from little string portions much faster than the String class
    - Comparable to Java's "StringBuffer" class
  - Example:
    - Method APPEND
      - Appends a new string chunk to the buffer
    - Method STRING
      - Renders the buffer to a single string and returns it

Classification Tree of Object Rexx, 6a

"Collection" Classes



Collection Classes, 1a

- Collection classes allow collecting Object Rexx objects
- The following methods are available
  - PUT or the synonym "[ ]=" **collects** (stores) an object

```
collectionObject ~PUT(object,index)
collectionObject ~"[]="(object,index) or
collectionObject[index] = object
```

  - AT or the synonym "[ ]" **retrieves** an object from a collection

```
collectionObject ~AT(index)
collectionObject ~"[]"(index) or
collectionObject[index]
```

Collection Classes, 1b

- Some collection classes allow supplying a list of objects to be put into the newly created collection object. In such a case the OF message (with the list of objects to be collected as its argument) is sent to the class instead of the NEW message, which creates an empty collection object.
- All collected objects can be iterated using the DO...OVER block statement
  - Processing loop

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

- Also one could use SUPPLIER objects for iterating over all of the collected objects (see below)

Collection Classes, 2

- One can arrange collection classes in two groups
  - Unordered collection
    - Collection classes ***without*** *a user defined index*
  - *Ordered* collection
    - Collection classes ***with*** *a user defined index*
- Ordered collection classes (*without* a user defined index)
  - Array
  - List
  - Queue, CircularQueue
  - (Stream)

Collection Classes, 3
Ordered Collection

- Array (1)
  - Array objects allow the storing the collected objects with a pre-defined numeric index, which must be a whole number starting with the value 1

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

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

Output:

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

Collection Classes, 4
Ordered Collection

- Array (2)
  - Array objects can possess arbitrary many dimensions
    - Attention! The needed memory is the Cartesian product of the maximum number of entries of each dimension

```
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
```

Output:

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

Collection Classes, 5
Ordered Collection

- List
  - List objects allow the storing of objets in the form of a list, i.e. in an ordered manner

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

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

Output:

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

Collection Classes, 6
Ordered Collection

- Queue, CircularQueue
  - Queue objects allow the storing of objects at the "head" (PUSH) or at the "tail" (QUEUE), i.e. in an ordered manner

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

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

Output:

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

Collection Classes, 7
Ordered Collection

- Stream
  - Stream objects allow the processing of streams of "lines" or "characters", being mostly files

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

  - With the help of the stream object **o** one is able to process the file "**test.dat**", by sending the stream object the appropriate messages, for instance: *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
```

Possible output of the contents of the above file:

```
a Stream:
[This is the first line.]
[]
[The previous line was empty, now the third one is processed!]
```

Collection Classes, 8
Unordered Collection

- Unordered collection classes
  - Directory - index (any string) associates *one* object only
  - Relation - index (any object) can associate *multiple* objects (ALLAT)
    - Bag - restriction: index and associated object are the same!
  - Table - index (any object) associates *one* object only
    - Set - restriction: index and associated object are the same!
  - (Stem - index (any string) associates *one* object only)
- There is no order in which the objects get collected
  - DO…OVER (also SUPPLIER objects) enumerate the collected objects in an arbitrary (unforeseeable) order!

Collection Classes, 9
Unordered Collection

- Directory
  - Directory objects allow the collecting of objects with a user defined index of type string (one object per index)

```
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 "Acronym 'WU':" tmpColl~wu || ", 'RGF':" tmpColl~rgf
SAY tmpColl~string || ":"
DO item OVER tmpColl
   SAY "[" || item || "]"
END
```

Output (order could be different):

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

Collection Classes, 10
Unordered Collection

- Relation
  - Relation objects allow the collecting of objects with a user defined index of any type (multiple objects per index possible)

```
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
```

Output (order could be different):

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

Collection Classes, 11
Unordered Collection

- Bag
  - Bag objects allow the collecting of objects with a user defined index of any type (multiple objects per index possible, index and object are the same, hence index can be left out)

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

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

Output (order could be different):

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

Collection Classes, 12
Unordered Collection

- Table
  - Table objects allow the collecting of objects with a user defined index of any type (one object per index)

```
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
```

Output (order could be different):

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

Collection Classes, 13
Unordered Collection

- Set
  - Set objects allow the collecting of objects with a user defined index of any type (one object per index, index and object are the same, hence index can be left out)

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

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

Output (order could be different):

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

Collection Classes, 14
Unordered Collection

- Stem
  - Stem objects allow the collecting of objects with a user defined index of type string (one object per index)

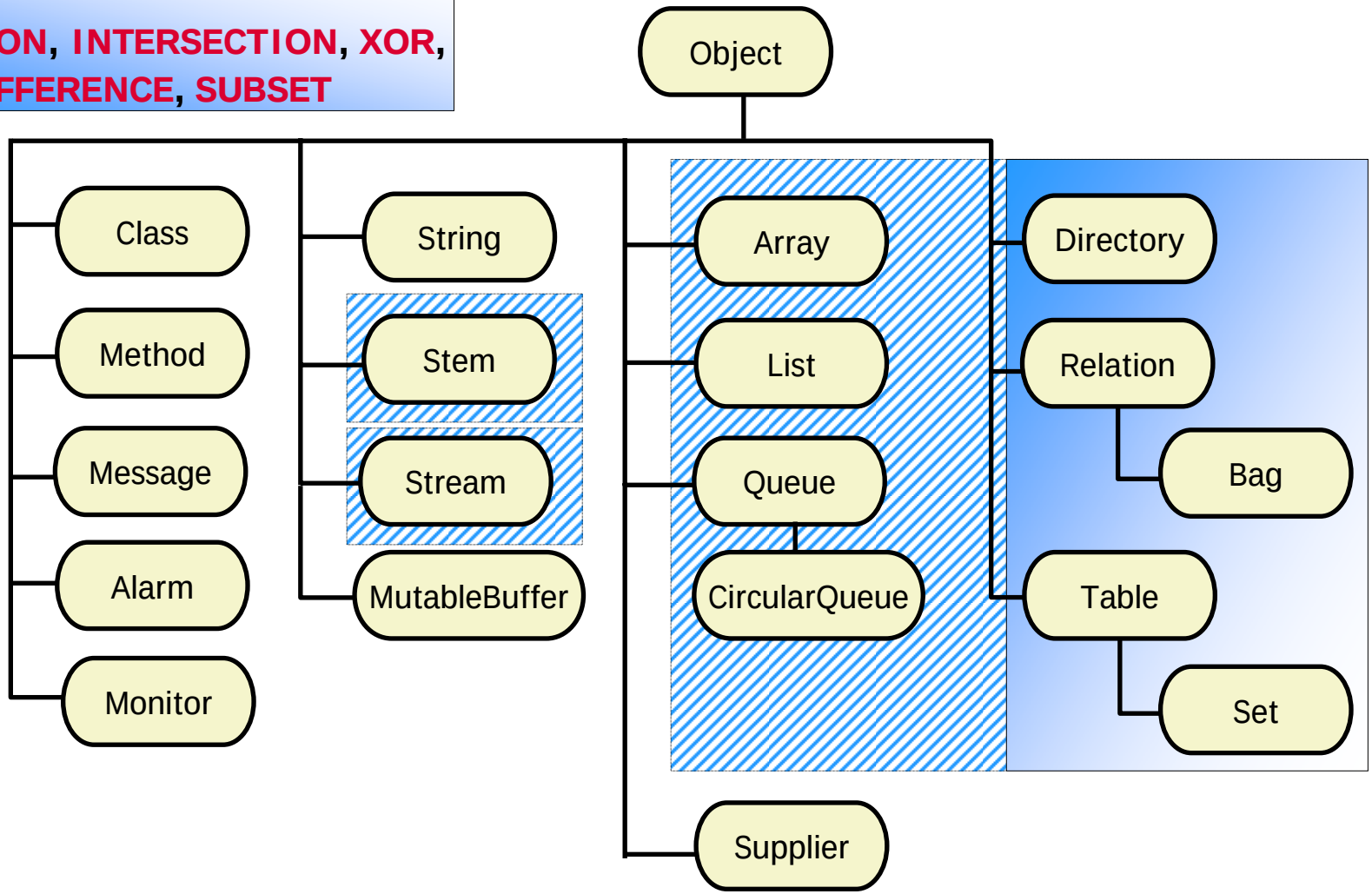
```
tmpColl = .stem ~new
tmpColl["a_index"] = "a"
tmpColl ~"[]="("b", "b_index")
tmpColl ~~PUT("b", "b_index" ) ~~PUT("c", "c_index")

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

Output (order could be different):

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

Classification Tree of Object Rexx, 6b

"Collection" Classes
with UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET



Collection Classes, 1
(UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET)

- Example 1 (two 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
```

Output (order could be different):

```
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
```

Collection Classes, 2
(UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET)

- Example 2 (Set and 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
```

Output (order could be different):

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

Collection Classes, 3
(UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET)

- Example 3 (Bag and 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
```

Output (order could be different):

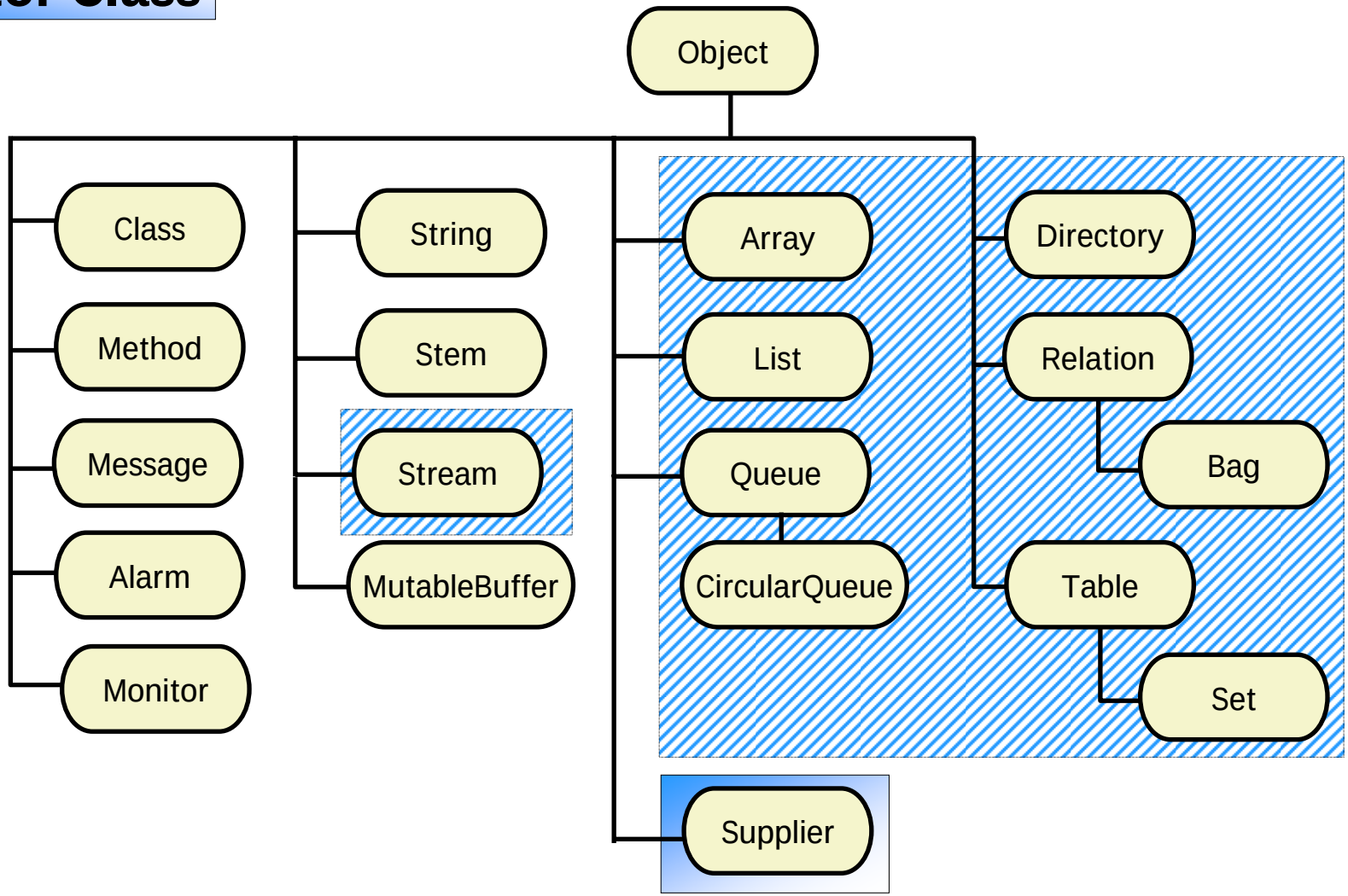
```
UNION        a Bag: [a] [b] [c] [b] [b]
INTERSECTION a Bag: [b]
XOR          a Bag: [a] [b] [c]
DIFFERENCE   a Bag: [a] [b]
1 - 0
```

Collection Classes, 4
(UNION, INTERSECTION, XOR, DIFFERENCE, SUBSET)

- Result is *always* an object of the same type as the receiving collection object
  - Argument of a setlike message can be an arbitrary collection object
    - If the argument collection object has no user defined index (Array, List, Queue, Stem, Stream) then the collection is turned into a bag collection containing the collected objects
  - Argument will be first converted to the type of the receiving collection before carrying out the operation

Classification Tree of Object Rexx, 6c

Iterator Class



Iterator Class, 1

- Supplier
  - Supplier objects allow enumerating all objects contained in a collection
    - A supplier object presents each collected object and supplies the index that object is associated with
    - As the index-object pair is returned by the supplier object, there is no need to know whether the underlying collection is one where the index is userdefined or not
  - The builtin collection classes possess a method SUPPLIER which returns the collection in the form of such a SUPPLIER object
  - Example code for enumerating all collected objects

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

Iterator Class, 2

- Example 1 (unordered collection: 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
```

Output (order could be different):

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

Iterator Class, 3

- Example 2 (ordered collection: 2-dimensional 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
```

Output:

```
an Array:
index [1,1] item [b]
index [1,2] item [c]
index [2,3] item [a]
index [4,5] item [b]
```

Iterator Class, 4

- Example 3 (unordered collection: 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
```

Output (order could be different):

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