Department of Information Systems
and Operations Management

Procedural and Object-oriented Programming 5b

Object Rexx Collection Classes

Business Programming 1 | **Business Programming 2**

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

Vienna University of Economics and Business ▪ Welthandelsplatz 1, D2-C ▪ A-1020 Vienna

Prof. Rony G. Flatscher

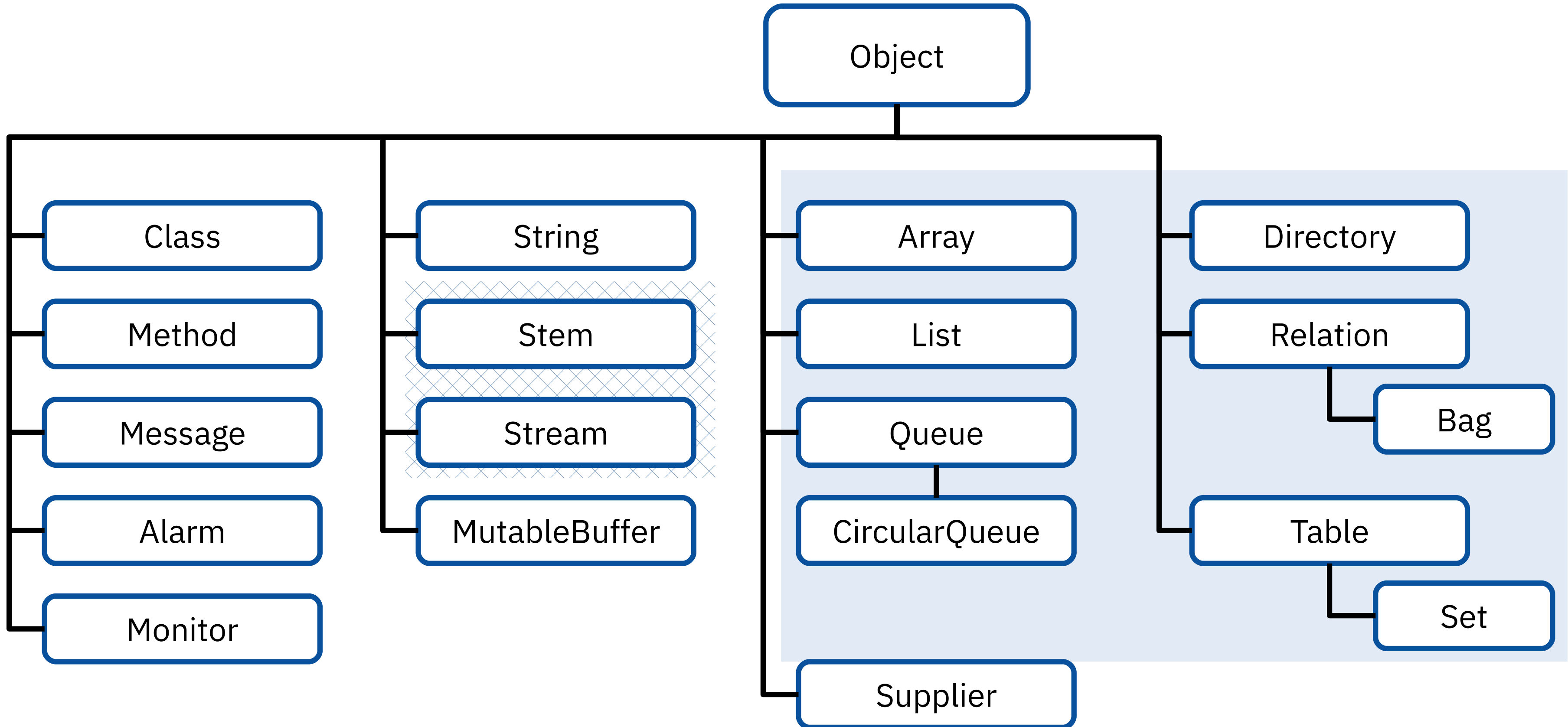
CC BY SA

Collection Classes, 1

- An overview of collection classes
  - Hint: ooRexx comes with more collection classes than get demonstrated here
- Common methods of collection classes
- Iterating over collected objects
  - Differences between ordered and unordered collection classes
- Collections with setlike methods

Cf. rexxref.pdf (5.3.2. Collection Class)

Collection Classes, 2



Cf. rexxref.pdf (5. Builtin Classes, 5.3.2. Collection Class)

CC BY SA

Collection Classes, 3

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

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

  - AT or the synonym "[ ]" retrieves an object from a collection by its index

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


Cf. rexxref.pdf (5.3. Collection Classes)

Collection Classes, 4

- Collection classes allow the use of the class method OF supplying a list of objects for the collection instead of the class method NEW.
- Iterating over the collection using a DO...OVER block statement

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

- Iterating over the collection using a SUPPLIER object (see below)

Cf. rexxref.pdf (5.3. Collection Classes)

Collection Classes, 5

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

Cf. rexxref.pdf (5.3. Collection Classes)

Collection Classes – Ordered Collection, 1

- **Array**
  - Array objects allow the storing of 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 i OVER tmpColl
   SAY "["i"]"
END
```

Output:

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


Cf. rexxref.pdf (5.3.6. Array Class)

CC BY SA

Collection Classes – Ordered Collection, 2

- **Array**
  - Array objects can possess arbitrary many dimensions
    - Hint: 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 i OVER tmpColl
   SAY "["i"]"
END
```

Output:

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


Cf. rexxref.pdf (5.3.6. Array Class)

Collection Classes – Ordered Collection, 3

- **List**
  - List objects allow the storing of objects (instances, values) in the form of a list, i.e. in an ordered manner

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

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

Output:

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


Cf. rexxref.pdf (5.3.11. List Class)

Collection Classes – Ordered Collection, 4

- **Queue, CircularQueue**
  - Queue objects allow the storing of objects (instances, values) 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 i OVER tmpColl
   SAY "["i"]"
END
```

Output:

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


Cf. rexxref.pdf (5.3.13. Queue Class, 5.3.8. CircularQueue Class)

CC BY SA

Collection Classes – Ordered Collection, 5

- **Stream**
  - Stream objects allow the processing of streams of "lines" or "characters"
    - With the help of the stream object one is able to process the file "myinput.txt", by sending the stream object the appropriate messages, for instance: OPEN, LINEIN (CHARIN), LINEOUT (CHAROUT), ARRAYIN, ARRAYOUT, CLOSE...

```
tmpColl = .stream ~new("myinput.txt")~~open
SAY "a" tmpColl~class~id":"
DO i OVER tmpColl
    SAY "["i"]"
END
tmpColl~close
```

```
myinput.txt:
Max
und
Moritz
haben
...
```

Output:

```
a Stream:
[Max]
[und]
[Moritz]
[haben]
[...]
```


Cf. rexxref.pdf (5.2.4. Stream Class)

Collection Classes – Ordered Collection, 6

- **Stream** and **Array**
  - The stream method ARRAYIN reads the stream's content into an array such that array methods can be employed (e.g. ITEMS, PUT, REMOVE, SORT, ...)

```
tmpColl = .stream ~new("myinput.txt")~~open
tmpArray = tmpColl~ArrayIn  -- read all lines into an array
tmpColl~close

SAY "a" tmpColl~class~id "becomes an" tmpArray~class~id","
SAY "the array has" tmpArray~items "items (lines):"
DO i OVER tmpArray
    SAY "["i"]"
END
```

```
myinput.txt:
Max
und
Moritz
haben
...
```

5 Items

Output:

```
a Stream becomes an Array,
the array has 5 items (lines):
[Max]
[und]
[Moritz]
[haben]
[...]
```


Cf. rexxref.pdf (5.2.4. Stream Class, 5.3.6. Array Class)

CC BY SA

Collection Classes – Unordered Collection, 1

- 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 (instances, values) in an arbitrary (unforeseeable) order!
    - Note: the loop variable in DO...OVER will supply the index with which an object is stored
      - One needs to use that index to fetch the object from the collection!

Cf. rexxref.pdf (5.3. Collection Classes)

Collection Classes – Unordered Collection, 2

- **Directory** (**StringTable**)
  - Directory objects allow the collecting of objects with a user defined index of type string

```
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 i OVER tmpColl
        SAY "["i"]"
END
```

Output:

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

order could be different

Cf. rexxref.pdf (5.3.9. Directory Class, 5.3.17. StringTable Class)

CC BY SA

Collection Classes – Unordered Collection, 3

- **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 i OVER tmpColl
   SAY "["i"]"
END
```

Output:

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

order could be different

Cf. rexxref.pdf (5.3.14. Relation Class)

CC BY SA

Collection Classes – Unordered Collection, 4

- **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 i OVER tmpColl
   SAY "["i"]"
END
```

Output:

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

order could be different

Collection Classes – Unordered Collection, 5

- **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 i OVER tmpColl
   SAY "["i"]"
END
```

Output:

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

order could be different

Cf. *rexxref.pdf (5.3.18. Table Class)*

Collection Classes – Unordered Collection, 6

- **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 i OVER tmpColl
   SAY "["i"]"
END
```

Output:

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

order could be different

Cf. *rexxref.pdf (5.3.15. Set Class)*

Collection Classes – Unordered Collection, 7

- **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 i OVER tmpColl
   SAY "["i"]"
END
```

Output:

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

order could be different

Cf. rexxref.pdf (5.3.16. Stem Class)

Collection Classes – Excursion, 1

- The MAKEARRAY method allows to transform any collection object into an **Array** object such that array methods can be employed (e.g. ITEMS, PUT, REMOVE, SORT, ...).

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

tmpArray = tmpColl~makeArray

SAY "a" tmpArray~class~id":"
DO i OVER tmpArray
   SAY "["i"]"
END
```

Output:

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


Cf. rexxref.pdf (5.3.16. Stem Class)

Collection Classes – Excursion, 2

- All collection classes have a MAKEARRAY method that allows to transform any collection object into an **Array** object such that array methods can be employed (e.g. ITEMS, PUT, REMOVE, SORT, ...).

```
tmpColl = .list~of("orange", "banana", "apple")

tmpArray = tmpColl~makeArray   -- turn collection into array
say "there are" tmpArray~items "items in the array (list)"
SAY "an" tmpArray~class~id":"
DO i OVER tmpArray
   SAY "["i"]"
END
say "... and in sorted order:"
DO i OVER tmpArray~sort        -- sort array
   SAY "["i"]"
END
```

Output:

```
there are 3 items in the array (list)
an Array:
[orange]
[banana]
[apple]
... and in sorted order:
[apple]
[banana]
[orange]
```


Cf. rexxref.pdf (5.3.16. Stem Class)

Collection Classes – Excursion, 3

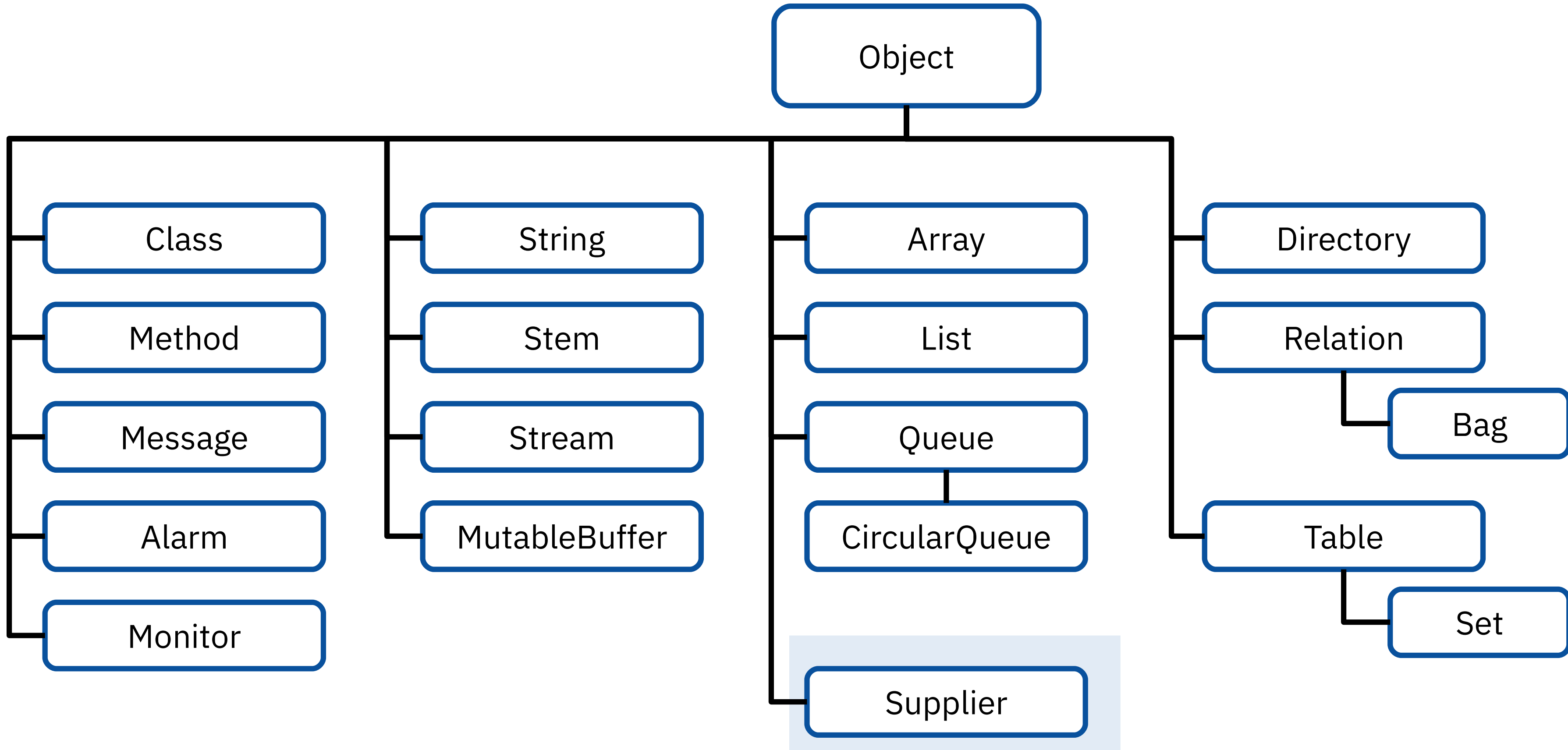
- **List, Array** and **Stream**
  - By default, the OPEN stream method opens the stream for reading and writing data
    - The read pointer is set to the first byte, the write pointer to the end of the file, such that writing to the file effectively appends it
    - Using the OPEN option `"replace"` causes the file's old content to be deleted
  - The stream method ARRAYOUT writes collected objects into a file.

```
tmpColl = .list~of("orange", "banana", "apple")

tmpArray = tmpColl~makeArray~sort    -- make & sort array

tmpStream = .Stream~new("myoutput.txt")~~open("replace")
tmpStream~ArrayOut(tmpArray)
tmpStream~close
```

```
myoutput.txt:
apple
banana
orange
```

Cf. rexxref.pdf (5.3.11. List Class, 5.2.4. Stream Class, 5.3.6. Array Class)

Iterator Class



Cf. rexxref.pdf (5. Builtin Classes)

CC BY SA

Iterator Class, 1

- **Supplier**
  - Supplier objects allow enumerating all objects contained in a collection
    - A supplier object presents each collected object together with its index
    - 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 user defined or not
  - All builtin collection classes possess a method SUPPLIER which returns a snapshot of the collection and returns it in the form of a SUPPLIER object
  - "Blueprint" code for enumerating all collected objects of a collection using the methods of its SUPPLIER object

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


Cf. rexxref.pdf (5.4.21. Supplier Class)

Iterator Class, 2

- **Supplier**
  - ooRexx 5 introduces the ability to loop over a collection object directly ("DO WITH ... OVER ...") assigning the index value and the item value to the defined loop variables in each iteration
  - "Blueprint" code for enumerating all collected objects (using implicitly the collection's SUPPLIER object)

```
DO WITH INDEX idx ITEM obj OVER tmpColl
   SAY "index ["idx"] item ["obj"]"
END
```

  - Note: this form of the DO instruction will send the *tmpColl* collection the message *supplier* and process the resulting SUPPLIER object

Cf. rexxref.pdf (2.4. DO, 5.4.21. Supplier Class)

Iterator Class, 3

- **Relation** (unordered collection)

```
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
```

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

SAY tmpColl~string":"


DO WITH INDEX idx ITEM obj OVER tmpColl
   SAY "index ["idx"] item ["obj"]"
END
```

Output:

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

order could be different

Cf. rexxref.pdf (2.4. DO, 5.4.21. Supplier Class, 5.3.14. Relation Class)

CC BY SA

Iterator Class, 4

- **2-dimensional Array** (ordered collection)

```
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
   idx = tmpSupp~INDEX
   SAY "index ["idx~makeString("L",",")"] item ["tmpSupp~ITEM"]"
   tmpSupp~NEXT
END
```

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

SAY tmpColl~string":"

DO WITH INDEX idx ITEM obj OVER tmpColl
   SAY "index ["idx~toString("L",",")"] item ["obj"]"
END
```

Output:

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

order will be always the same!

Cf. rexxref.pdf (2.4. DO, 5.4.21. Supplier Class, 5.3.6. Array Class)

CC BY SA

Iterator Class, 5

- **Set** (unordered collection)

```
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
```

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

SAY tmpColl~string":"


DO WITH INDEX idx ITEM obj OVER tmpColl
   SAY "index ["idx"] item ["obj"]"
END
```

Output:

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

order could be different

Cf. rexxref.pdf (2.4. DO, 5.4.21. Supplier Class, 5.3.15. Set Class)

CC BY SA

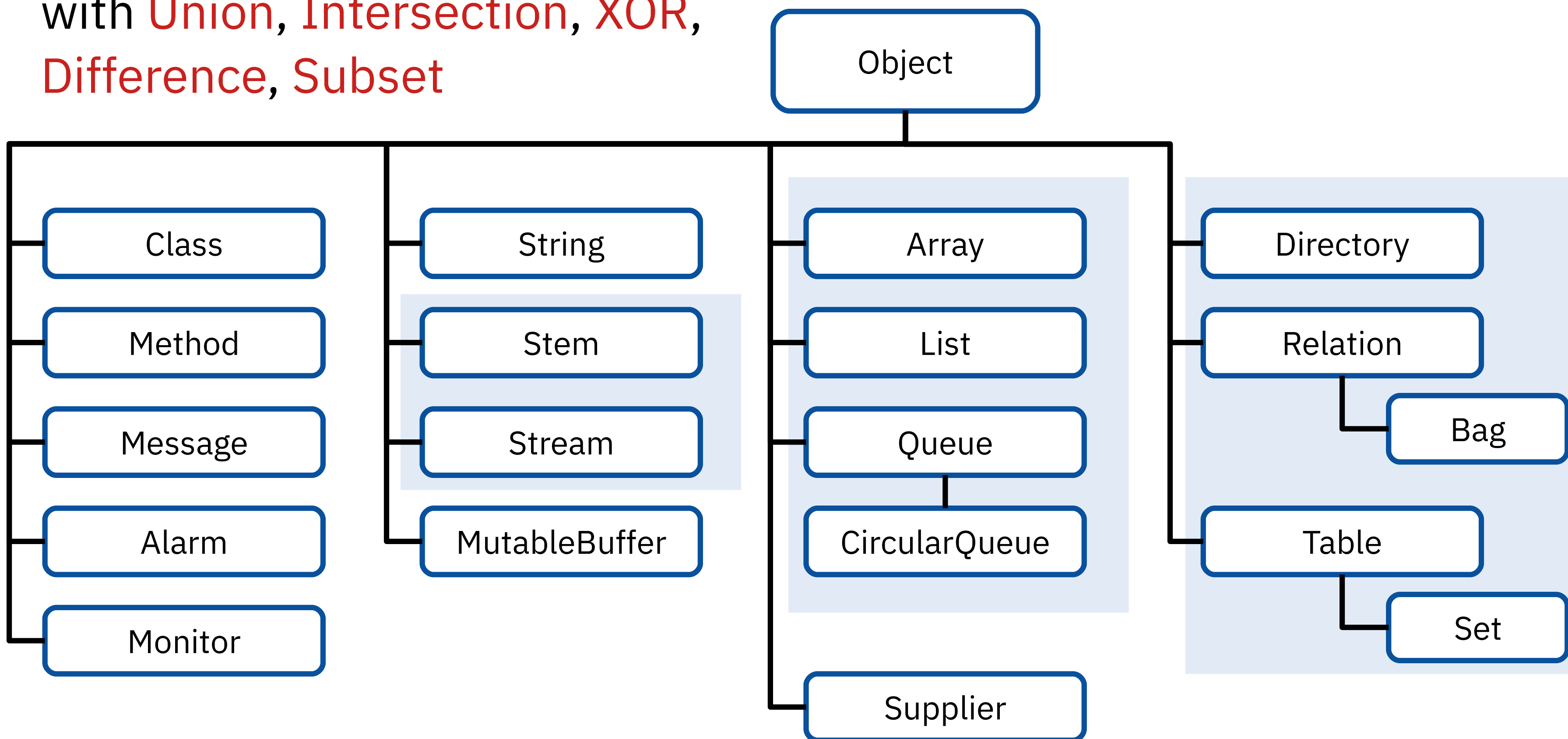
Setlike Operations

- *Union* of two collections
  - Result includes all items from both, the receiver and the argument collection
- *Intersection* of two collections
  - Result includes all items of the receiver collection that are also in the argument collection
- *XOR* of two collections (exclusive or)
  - Result includes all items that are contained only in the receiver, but not in the argument collection and vice versa
- *Difference* (remove items from receiver collection)
  - Result removes all items from the receiver collection that are in the argument collection
- *Subset*
  - Returns *.true*, if the receiver collection is a subset of the argument collection

Cf. rexxref.pdf (5.3.2. Collection Class)

Collection Classes: Union, Intersection, XOR, Difference, Subset

- with Union, Intersection, XOR, Difference, Subset



Cf. rexxref.pdf (5. Builtin Classes, 5.3.2. Collection Class)

Collection Classes: Union, Intersection, XOR, Difference, Subset, 1

REXX™

- Two **Bag** collections

```
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
  .output~CHAROUT( title tmpColl~string": ")
  DO i OVER tmpColl
     .output~CHAROUT("["i"] ")
  END
  .output~say
```

Output:

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

order could be different

Cf. rexxref.pdf (5.3.7. Bag Class)

CC BY SA

Collection Classes: Union, Intersection, XOR, Difference, Subset, 2

- A **Set** and a **Bag** collection

```
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
  .output~CHAROUT( title tmpColl~string": ")
  DO i OVER tmpColl
     .output~CHAROUT("["i"] ")
  END
  .output~say
```

Output:

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

order could be different

Cf. rexxref.pdf (5.3.15. Set Class, 5.3.7. Bag Class)

Collection Classes: Union, Intersection, XOR, Difference, Subset, 3

- A **Bag** and a **Set** collection

```
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
  .output~CHAROUT( title tmpColl~string": ")
  DO i OVER tmpColl
     .output~CHAROUT("["i"] ")
  END
  .output~say
```

Output:

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

order could be different

Cf. rexxref.pdf (5.3.15. Set Class, 5.3.7. Bag Class)

Collection Classes: Union, Intersection, XOR, Difference, Subset, 4

- Result is always an object of the same type as the receiving collection object
  - Argument of a setlike message can be any collection object
  - Argument will be first converted to the type of the receiving collection before carrying out the operation
    - If the argument collection object has no user defined index (i.e. it is an ordered collection like e.g. Array, CircularQueue, List, Queue, Stem, Stream) then the collection is converted into a Bag collection by putting all its collected objects into it

Cf. rexxref.pdf (5.3.2. Collection Class)

CC BY SA