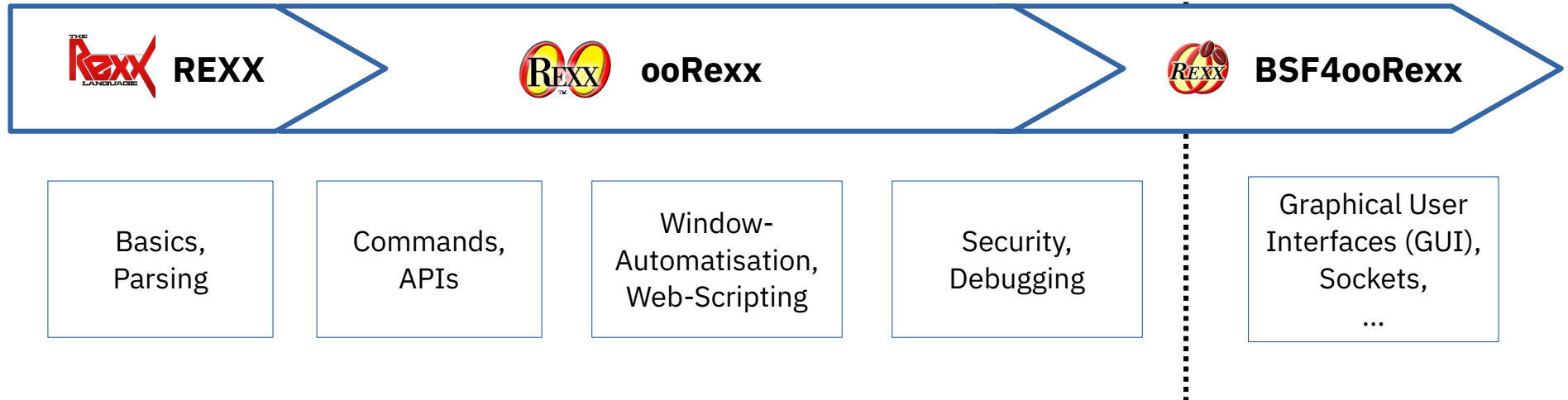


# Procedural and Object-oriented Programming 5

Defining Classes ("CLASS" Directive), Defining Methods ("METHOD" Directive),  
Object REXX Classes

## Business Programming 1

## Business Programming 2



# 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 instances of the same type
  - Possess all the same attributes (constitute the datastructure as defined in the class) and operations ("methods of the 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 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`



# 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, 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")
  - Execution (resolution) of messages: from left to right



# 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
```



# 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, that are defined before any class directive ("floating attributes")



- Creating (constructing) a new object (value, instance) 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**"
  - Always invoke the **INIT** method of the superclass!

## Constructor: Method **INIT**



```
/**/  
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  
::ATTRIBUTE firstName  
::ATTRIBUTE familyName  
::ATTRIBUTE salary  
::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 destructed by the garbage collector by sending it the **UNINIT** message.
- The **UNINIT** method is therefore called "***destructor***"



## Destructor: Method UNINIT



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

### Output:

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

Sequence can  
be different on  
different runs!

# 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 (attributes and methods) of all superclasses up to and including the root class
    - Subclasses **specialize** in one way or the other the superclass(es)
      - "Defining of differences": simplifies the definition of subclasses
  - 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



- In method routines ooRexx sets the following two variables which are therefore *always available in methods*
  - **super**
    - Always contains a reference to the immediate superclass
    - Allows redirecting the search for methods to the immediate 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 with the message name starts



# Example "Dog", 1



- Problem description
  - "Special Interest Group (SIG) Dog Sanctuary"
    - 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  
::ATTRIBUTE Name  
::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
  - 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 shielded 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 name *XYZ* is created
- The following options are available for this directive
  - **PRIVATE, PUBLIC**
    - Optional, default value: **PRIVATE**
  - **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, 1

## Implementing "Vehicle", "RoadVehicle", "WaterVehicle"

```
/**/  
.RoadVehicle ~new("Truck") ~drive  
.WaterVehicle ~new("Boat") ~swim  
  
::CLASS Vehicle  
::ATTRIBUTE name  
::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, 2

# Implementing "AmphibianVehicle" Using Multiple Inheritance



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

::CLASS      Vehicle
::ATTRIBUTE  name
::METHOD     INIT
             self~name = ARG(1)
::METHOD     tsk
             SAY self~name": 'tsk!'"

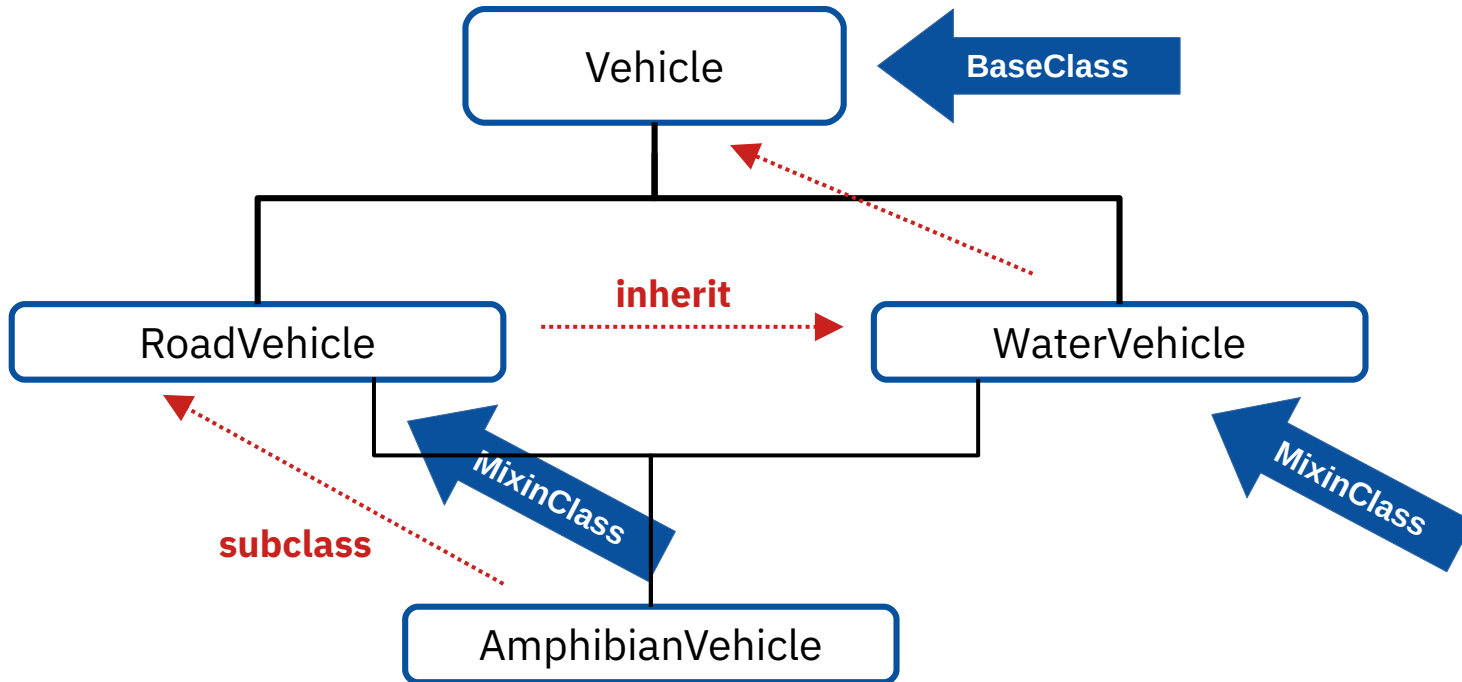
::CLASS      RoadVehicle MIXINCLASS Vehicle
::METHOD     drive
             SAY self~name": 'I drive now...'"

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

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

Output:

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



# ::METHOD Directive, 1



- This directive causes the interpreter to create a method
  - **::Method** mmm
    - A method with the identifier "MMM" is created
- The following options are available for this directive
  - **ATTRIBUTE**
    - Optional, if supplied the interpreter creates **two** methods:
      - A get ("getter") method "MMM" (returns the attribute's value) like this

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

- A set ("setter") method "MMM=" (sets the attribute's value to the supplied argument) like this

```
::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, 2



- The following options are available for this directive (continued)
  - PRIVATE, PUBLIC
    - Optional, default value: PUBLIC
    - If set to PRIVATE then the message can only be sent as: **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, UNPROTECTED
    - Optional, default value: UNPROTECTED
    - If set to PROTECTED then access to this method can be supervised with the help of the Object REXX *Security Manager*



# ::ATTRIBUTE Directive



- **::ATTRIBUTE** *mmm* [**GET|SET**]
  - This directive is equivalent to "**::METHOD** *mmm* **ATTRIBUTE**" and causes the interpreter to create the following two methods by default:
    - A getter method named "**MMM**" and
    - A setter method named "**MMM=**"
  - If the option **GET** is given, then only the getter method gets created
  - If the option **SET** is given, then only the setter method gets created





# ::CONSTANT Directive



- **::CONSTANT** NAME VALUE

- This directive creates a class and an instance method named **NAME** which always returns **VALUE**

```
say "pi:" .MyClass~pi "(from class)"
o=.MyClass~new
say "pi:" o~pi "(from instance)"

::CLASS MyClass
::CONSTANT pi 3.141592653589793238462643383279502884197
```

Output:

```
pi: 3.141592653589793238462643383279502884197 (from class)
pi: 3.141592653589793238462643383279502884197 (from instance)
```



# ::RESOURCE Directive, 1



- **::RESOURCE NAME**
  - By default this directive needs as delimiter the string "**::END**" starting at the first column of one of the following lines
    - The end marker string can be changed using the **END delimiter** option of the directive
  - All lines between the start and the end of the directive will be stored in an array
  - This array will be stored using **NAME** as its index in the **.RESOURCES TextTable**
- Fetching a resource in the program
  - Send **NAME** as the message to the **.RESOURCES TextTable**
    - Returns an array of text lines representing the named resource
      - To turn an array back into a plain string, send the array the **makeString** or the **toString** message
      - The **SAY** keyword statement will automatically request the string representation



# ::RESOURCE Directive, 2



- The **RESOURCE** directive makes it easy to define and use multi line strings
  - No need to enquote and concatenate strings spanning multiple lines
- Resources can be used among other things for e.g.
  - Multiline SQL queries
  - XML/HTML chunks to serve client requests in web server applications
  - Any kind of multiline text
  - *Base64* encoded binary data like pictures, sound, cryptographic keys, ...
    - Cf. the methods of the **String** class named `encodeBase64` and `decodeBase64`



# ::RESOURCE Directive, 3



```
say "resource named 'info':"
say .resources~info

::resource info
*****
*
* This is the secret:
*
*   Eat an apple a day to keep the doctor away! ;)
*
*****
::END
```

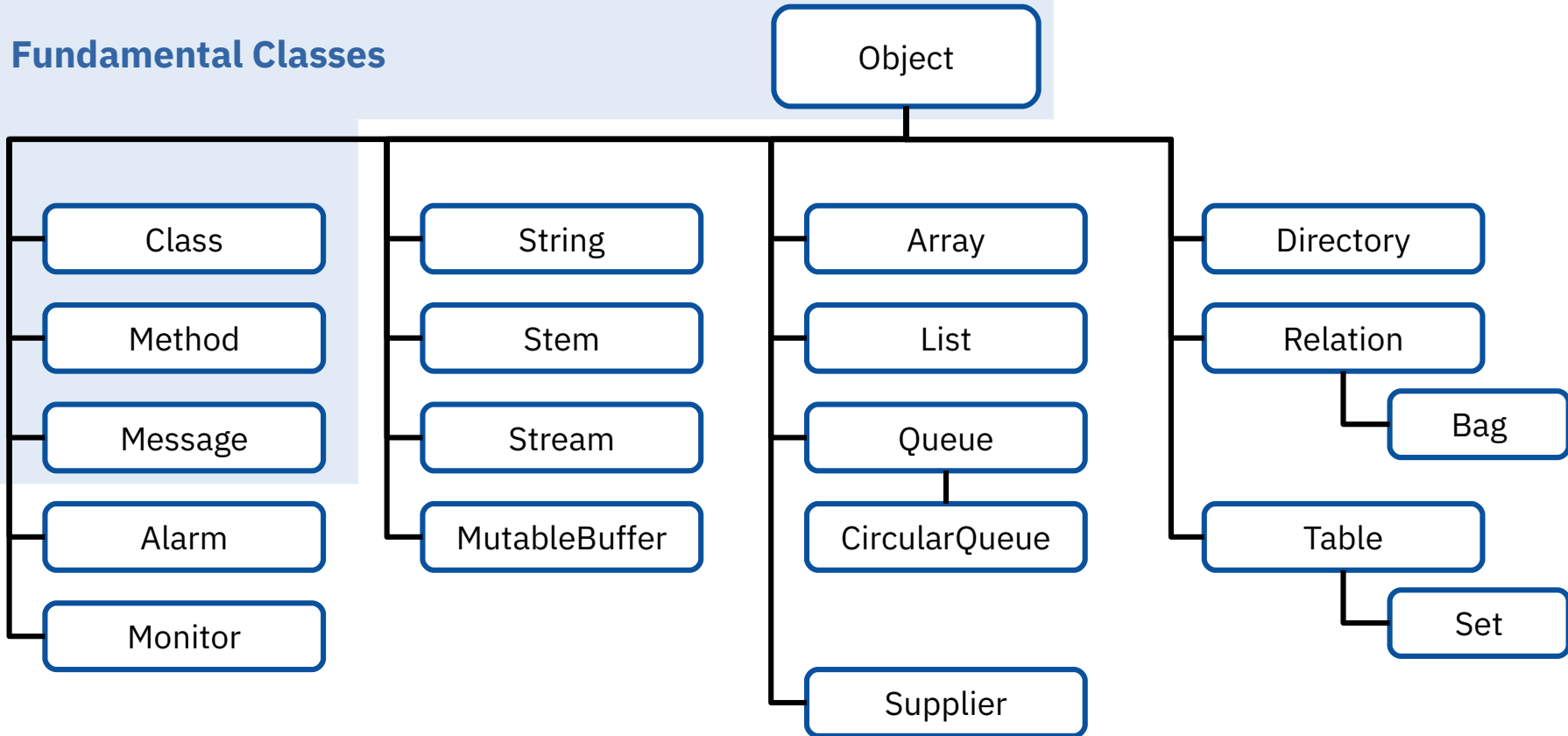
Output:

```
resource named 'info':
*****
*
* This is the secret:
*
*   Eat an apple a day to keep the doctor away! ;)
*
*****
```



# Fundamental Classes, 1

## Fundamental Classes



# Fundamental Classes, 2

- **Object**

- Methods and attributes are available to all instances (**objects**, **values**) of any Rexx class
  - Example: method **INIT**
    - Constructor, initializes a newly created object

- **Class**

- Interpreter creates an instance of this class ("**class object**") for each **::CLASS** directive
  - Example: method **ID**
    - Returns the name (the "**id**entification") of the class object
  - Example: method **NEW**
    - Creates a new instance (object, value) of the class, sends it the **INIT** message and returns it

# Fundamental Classes, 3

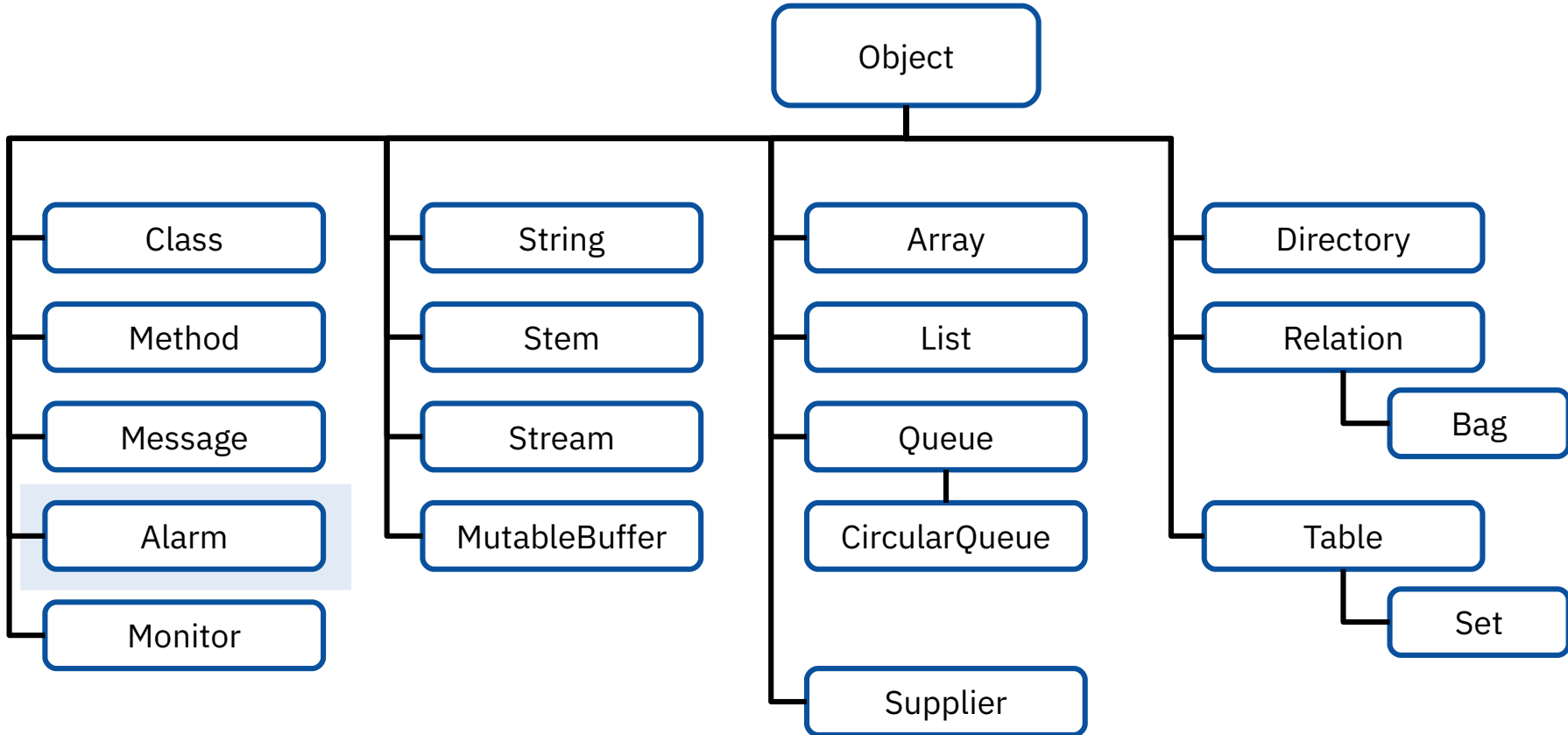
- **Method**

- 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 ("message object")
  - Example: method `SEND`
    - Sends (transmits, dispatches) the message to the object and waits until it got processed

# Alarm Class, 1





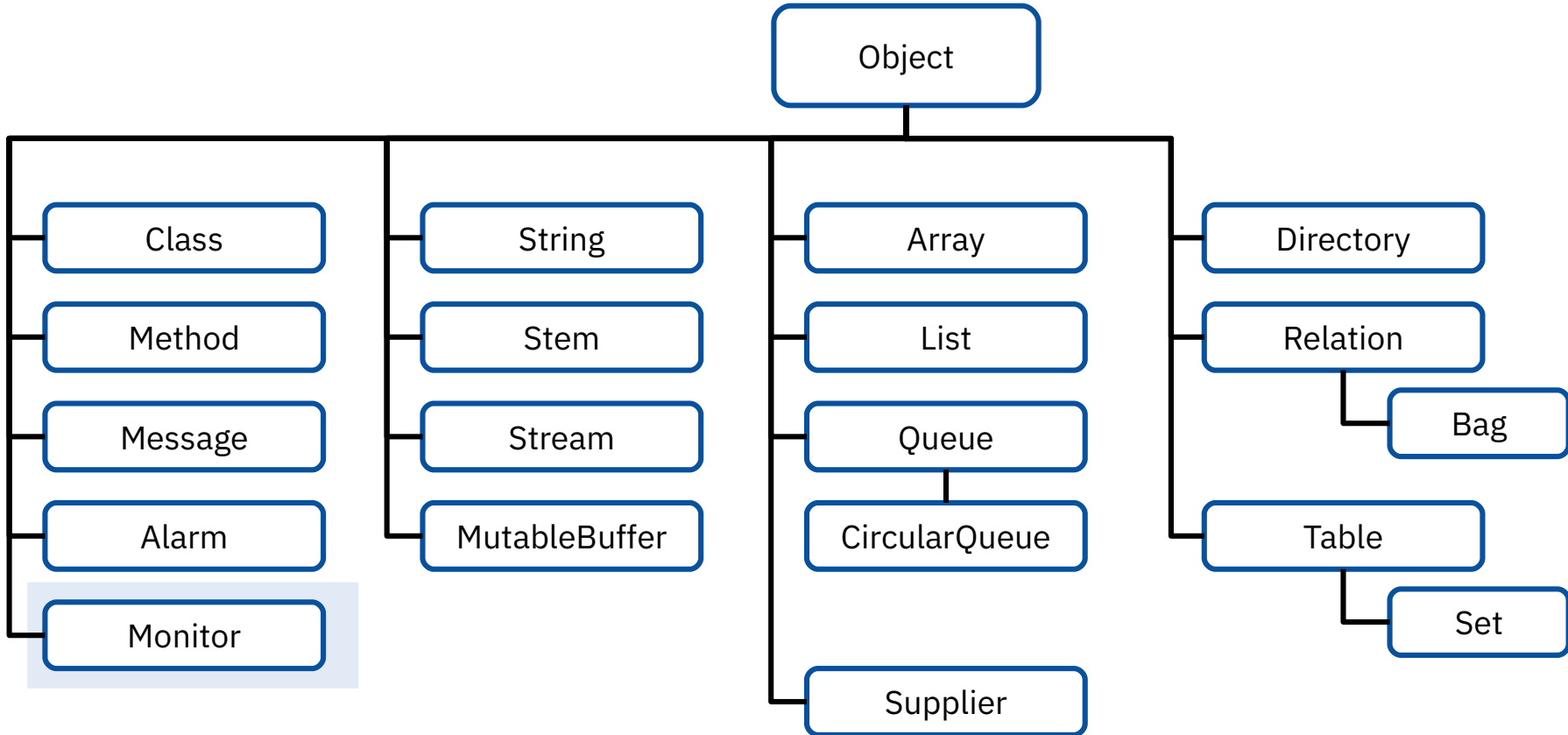


- **Alarm**

- 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 date and time
- Example: method **CANCEL**
  - Cancels an alarm object, the pending message will not be dispatched



# Monitor Class, 1

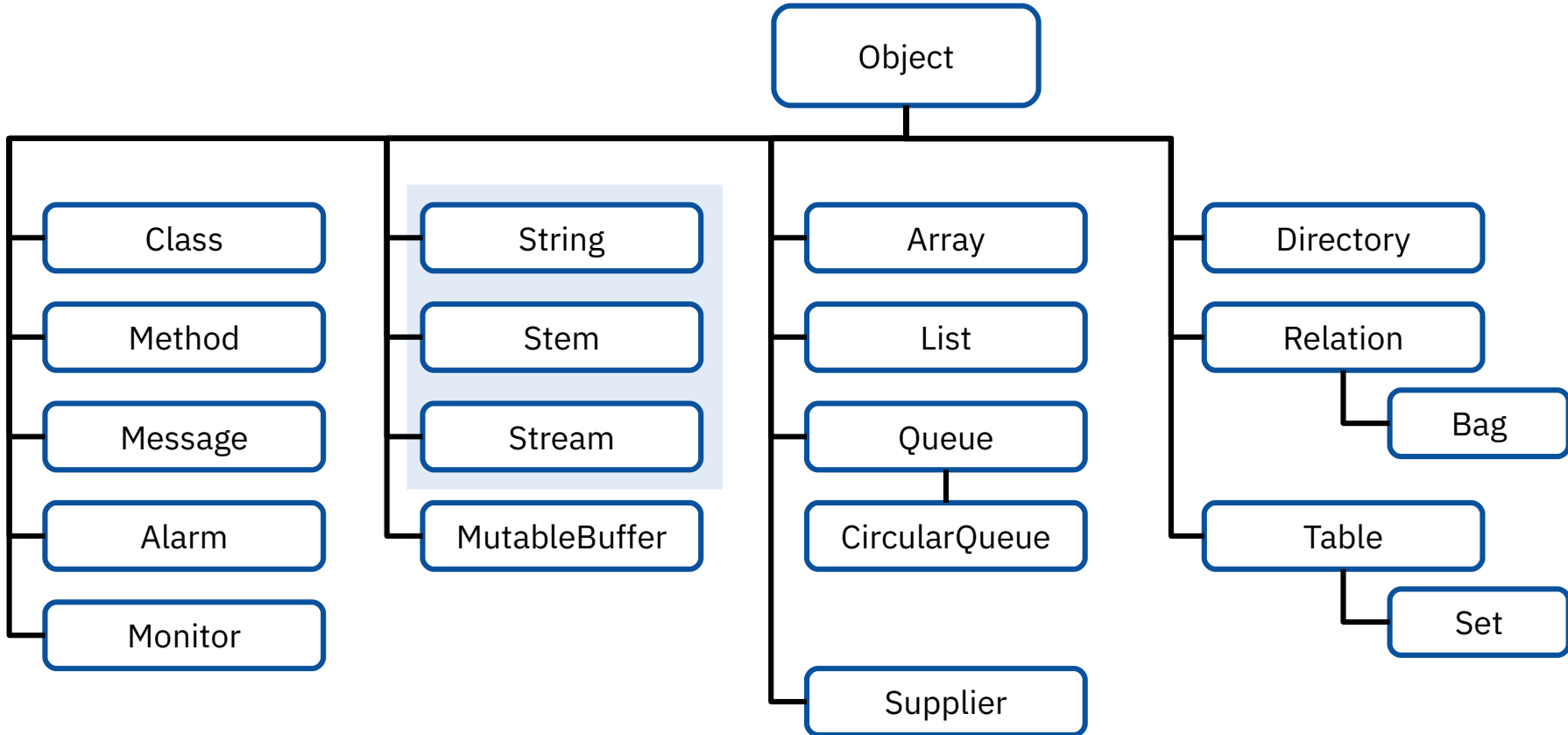




- **Monitor**
  - Monitor objects allow the monitoring of messages sent to objects
    - Example: method **CURRENT**
      - Returns the currently monitored object
    - Example: method **DESTINATION**
      - Allows to change the destination of the monitor
  - Hint: ooRexx uses monitors in its **.local** environment directory
    - **.input** monitors **.stdin**, the ooRexx *stdin* stream object
    - **.output** monitors **.stdout**, the ooRexx *stdout* stream object
    - **.error** monitors **.stderr**, the ooRexx *stderr* stream object



# “Classic REXX” Classes, 1



## “Classic REXX” Classes, 2

- **String**

- 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, 3

- **String**

- 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, 4

- **Stem**

- 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
```

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

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

```
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]
```

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

# “Classic REXX” Classes, 5

- **Stem**

- 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, 6

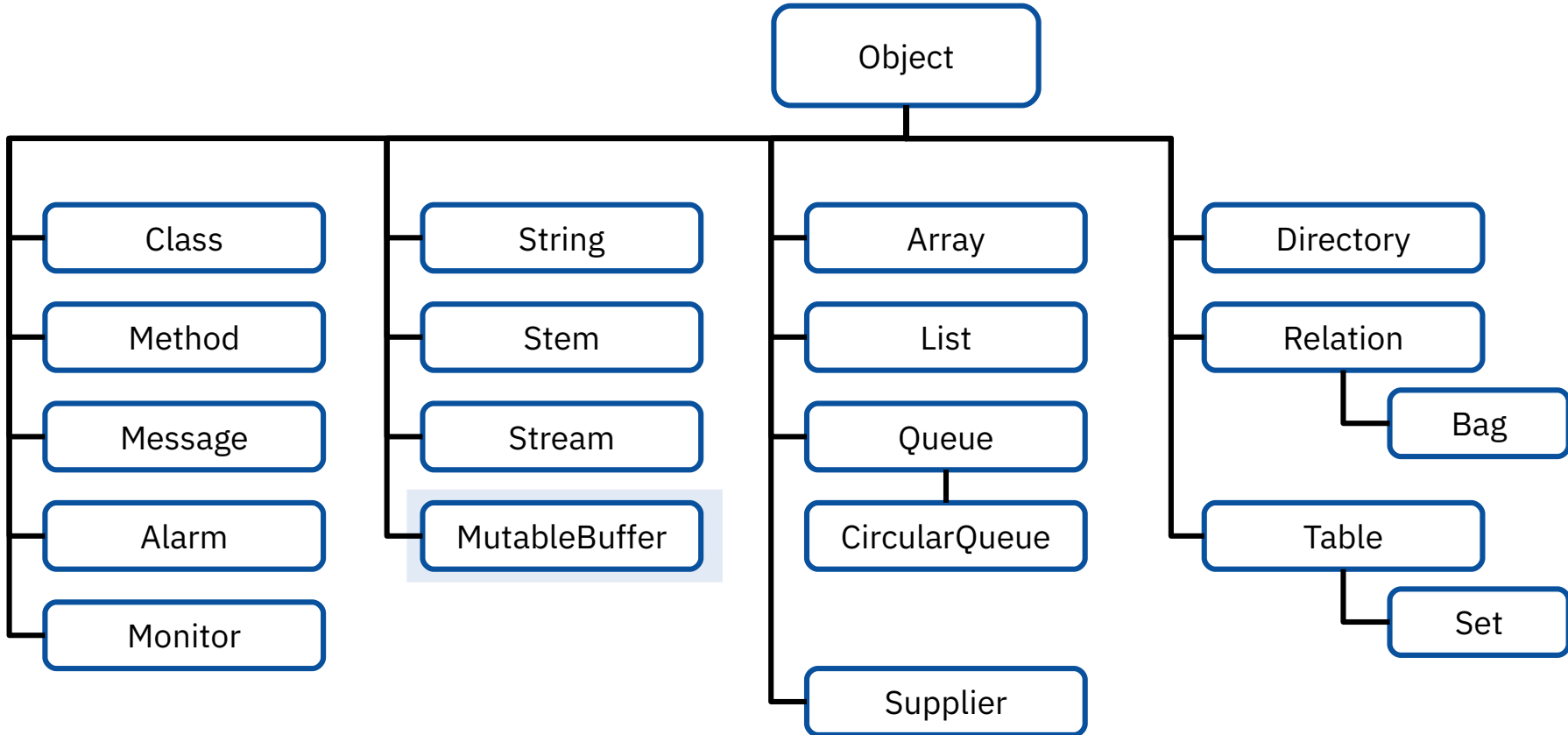
- **Stream**

- Stream objects allow working with files (and communication devices)
  - 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

# MutableBuffer Class, 1



## MutableBuffer Class, 2

- **MutableBuffer**
  - Class that allows to create a buffer from many little strings quickly
    - Comparable to Java's *StringBuffer* or *StringBuilder* classes
  - Example methods
    - Method **APPEND**
      - Appends a string chunk to the buffer
    - Method **STRING**
      - Renders the current buffer as a single string object and returns it