



Object Oriented Programming in

"COBOL 2000"

(Update as of 4/99)

Artur Reimann

Fujitsu Software Corporation
Development Tools Group
3055 Orchard Drive
San Jose, California 95134-2022

Copyright Fujitsu Software Corporation

Contents

<i>Object Orientation</i>	4
Objects	5
Classes	6
Classes and Objects in COBOL	7
Class Definition	8
Factory Definition	8
Object Definition	8
Method Definition	9
<i>Class Hierarchy and Inheritance</i>	10
Inheritance	10
Multiple Inheritance	11
<i>Accessing Objects and Methods</i>	12
Object References	12
Messages	13
Messages in COBOL	13
<i>Creating Objects</i>	15
COBOL Class BASE	15
Creating Objects in COBOL	15
<i>Conformance and interfaces</i>	16
Interfaces	16
Conformance	16
COBOL INTERFACE definition	16
Polymorphism	17
<i>Other OO Features in COBOL</i>	18
Object Modifier	18
Property	18
Generic Classes	18
<i>Exception Handling</i>	20
<i>Examples</i>	21
Example 1	21
Example 2 (with attribute)	22
Example 3 (with inheritance)	24

Object Orientation

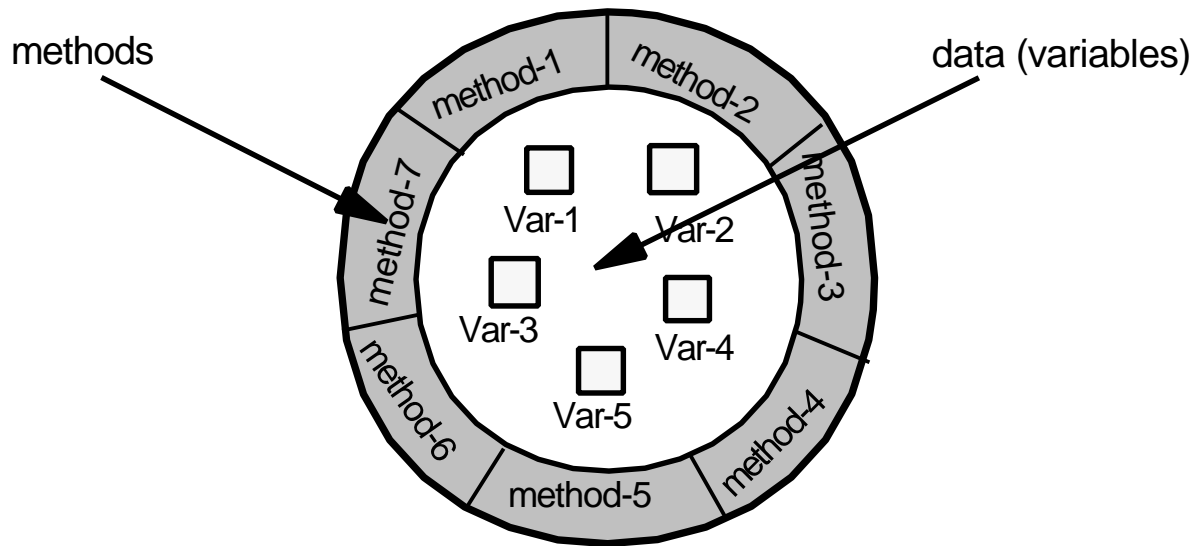
Object Orientation is the big new programming paradigm of these days. "COBOL 2000" (see separate document for more details and other features planned for the new standard) will provide full support for the constructs of Object Oriented Programming (OOP). Superficially, the role of OOP in COBOL 2000 is similar to that of Structured Programming in COBOL 85. There is, however, a very important difference: While Structured Programming was meant to be an *alternative* to traditional, "spaghetti-oriented" programming, believed to be simply a better programming style, OOP will not replace conventional programming, but will be used just for specific programming tasks, for which it is better suited. Traditional COBOL programming will not go away, but will be complemented by new language features for defining *classes* and *objects*.

This document describes the new language elements in a very general way. For better understanding, it also gives an overview of the *concepts* of OOP. Please understand, that this can be in no way a complete presentation of this complex matter. Some knowledge of OOP in general must therefore be assumed.

Some of the fundamental concepts of Object Orientation are the following:

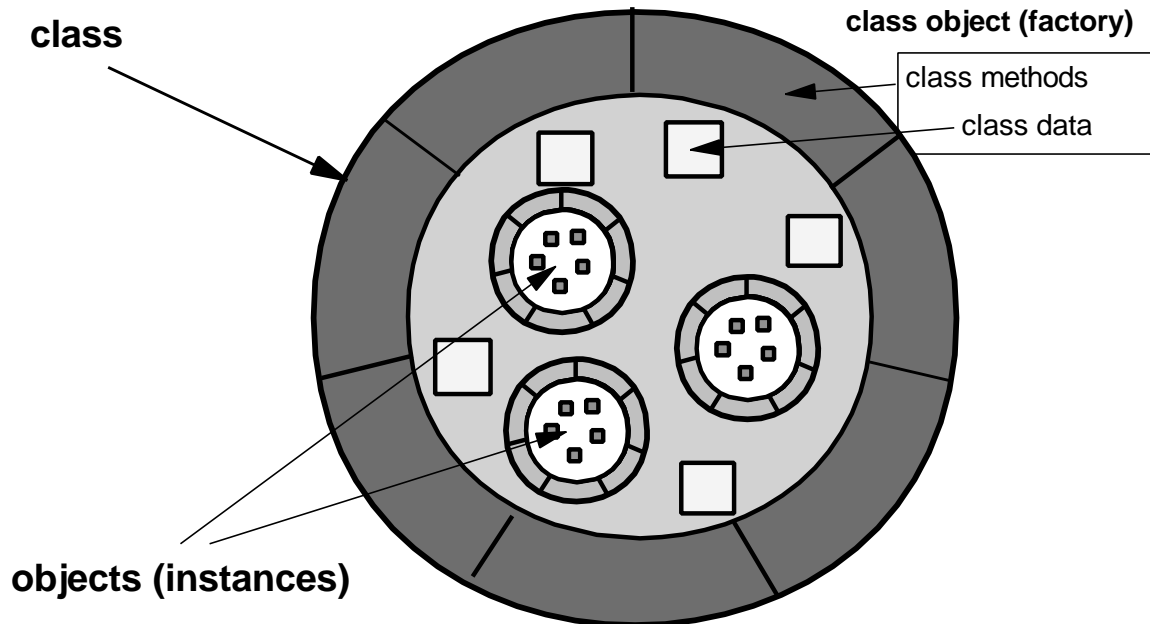
- Objects
 - Data together with methods for these data form objects
- Data encapsulation
 - Objects 'hide' their data
- Classes
 - Objects of same kind form classes
- Inheritance
 - Sub-classes 'inherit' data and methods of their super-classes
- Polymorphism
 - Methods with same name in different classes have different effect

Objects



This picture is a schematic view of an object. An object consists of data (variables, attributes), describing the state of the real object at any given time, and methods (procedures, programs), needed to access and to manipulate these data. The data are "encapsulated" in the object, i. e. they are normally not visible and certainly not changeable from outside the object. This is only possible through methods associated with the object.

Classes



Objects of the same kind are organized in "classes", schematically shown in the above picture. A class is something like a "prototype" of its objects. In OOP, you don't describe individual objects, but only their class.

An individual object, or "instance", will be created by the class, using a method that is associated with the class just like object methods are associated with the object. Thus, a class can be thought of a special object with its own methods and its own class data. It is called "factory object", because it is primarily used to *create* instances.

Classes and Objects in COBOL

The COBOL syntax for a class definition uses a similar concepts as the concept of Nested Programs, introduced in the COBOL 85 standard. The class itself has the role of the outermost program, factory and object definitions are nested inside the class definition, and method definitions in turn nested in the factory and object definitions.

```
Id Division.
Class-Id. class-name-1 Inherits class-name-2 ... .
...
Environment Division.
Configuration Section.
Repository.
    Class class-name-3 as "external-class-name-3"
    ...
    Id Division.
    Factory.
    Environment Division.
    ...
    Data Division.
    Working-Storage Section.
    ...
    Procedure Division.
        {class methods}
    End Factory.

    Id Division.
    Object.
    Environment Division.
    Data Division.
    ...
    Procedure Division.
        {object methods}
    End Object.
End Class class-name-1.
```

Example 1. Class Definition

```
Id Division.
Method-Id. method-name-1 ...
...
Data Division.
Working-Storage Section.
...
Linkage Section.
...
Procedure Division [Using ...] [Returning ...] .
...
    Invoke object-reference method [Using ...] [Returning ...]
...
Exit method.
End Method method-name-1.
```

Example 2. Method Definition

Class Definition

- Format:

```
[ID DIVISION]          (parameterized classes not covered here, see below)
CLASS-ID. class-name-1 [ AS literal-1 ]
      INHERITS { class-name-2 ... }.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
      [ { CLASS class-name-3 [ AS "external-class-name-3" ] ... ].
      [ { INTERFACE int-name-1 [ AS "external-int-name-1" ] ... ].
[ { factory definition } ]
{ object definition }
END CLASS class-name-1.
```

Factory Definition

- The factory object contains data and methods associated with the class, for example for creating objects
- Each class has only one factory object
- Environment Division contains only Input-Output Section
- Factory data and methods are accessible from all object methods of that class
- defined by:

```
[ID DIVISION.]
FACTORY [IMPLEMENTS interface-name-1].
[Factory Environment Division]
[Factory Data Division]
[PROCEDURE DIVISION.
[{factory methods}...]]
END FACTORY.
```

- IMPLEMENTS clause indicates that the interface of this factory object conforms to *interface-1* (see COBOL INTERFACE definition on page 16)

Object Definition

- The object definition contains data and methods associated with the objects, i.e. the instances of the class
- Environment Division contains only Input-Output Section.
- Defined by:

```
[ID DIVISION.]
OBJECT [IMPLEMENTS interface-name-1].
[Object Environment Division]
[Object Data Division]
[PROCEDURE DIVISION.
```

```
[{object methods}...]
END OBJECT.
```

- IMPLEMENTS clause indicates that the interface of this object conforms to *interface-1* (see COBOL INTERFACE definition on page 16)

Method Definition

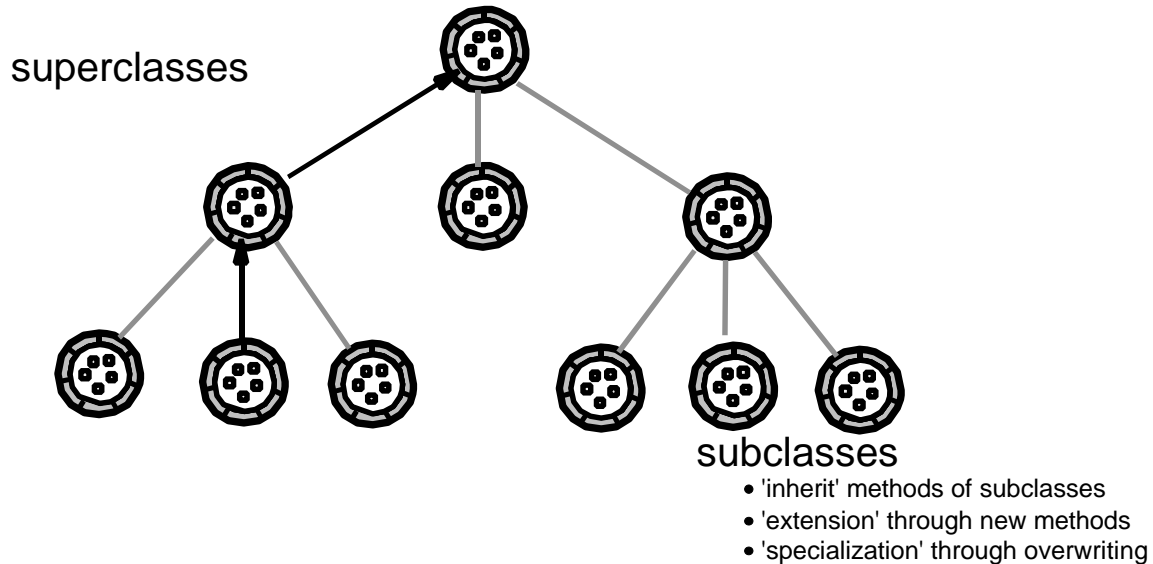
- Method definitions are contained in the Procedure Division of the Factory or the Object
- The method definition contains object procedures and data associated with the methods.
- Defined by:

```
[ID DIVISION.]
METHOD-ID. { method-name-1 [AS literal-1]
             { [GET]
               [SET] } PROPERTY property-name-1 } [ OVERRIDE ].
[Method Environment Division]
[Method Data Division]
[PROCEDURE DIVISION [USING {data-name-1}...]
                    [RETURNING data-name-2]].
    {statements}...
END METHOD method-name-1.
```

- Property (details: see Property on page 18)
 - property method
- Override
 - overrides inherited method

Class Hierarchy and Inheritance

Inheritance

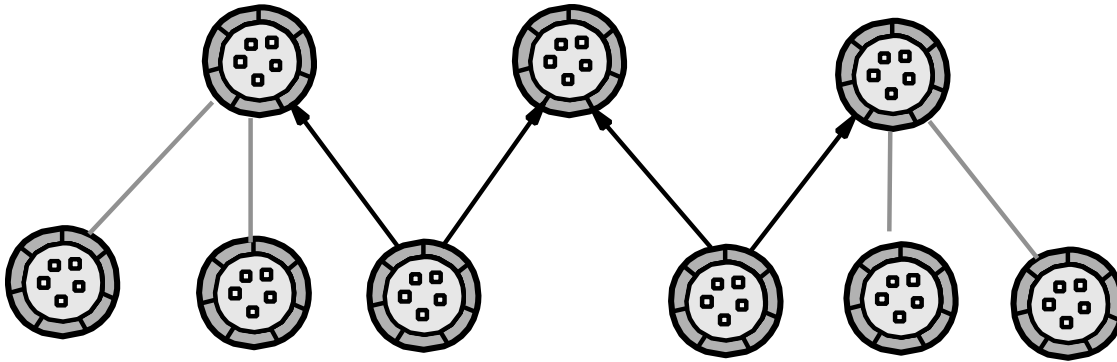


CLASS-ID *class-name-1* Inherits *class-name-2*

Separate classes can have common attributes, which means common data and method definitions. For example, convertibles and sedans are both cars and have many commonalities. Therefore, it makes sense to define a class "car" with all the common data and methods. Such a class would be called a "superclass" of the classes "convertible" and "sedan". The common data and methods are defined only for the superclass and inherited by the "subclass".

It is possible to define a specialized method that is to be used in place of an inherited method with the same name, as long as its "signature", i. e. the set of parameters and their properties, "conforms" to the inherited method (see "Conformance" on page 16).

Multiple Inheritance



```
CLASS-ID. class-name-1 Inherits class-name-2 class-name-3...
```

COBOL 2000 supports the concept of "multiple inheritance", which means that a class can have several superclasses. Thus the subclass inherits all the methods from all the superclasses.

Accessing Objects and Methods

Object References

The individual object (instance) has no name. It is addressed by an object reference. An object reference is allocated when an object is created, and returned to the invoking program or method.

- Universal object references:

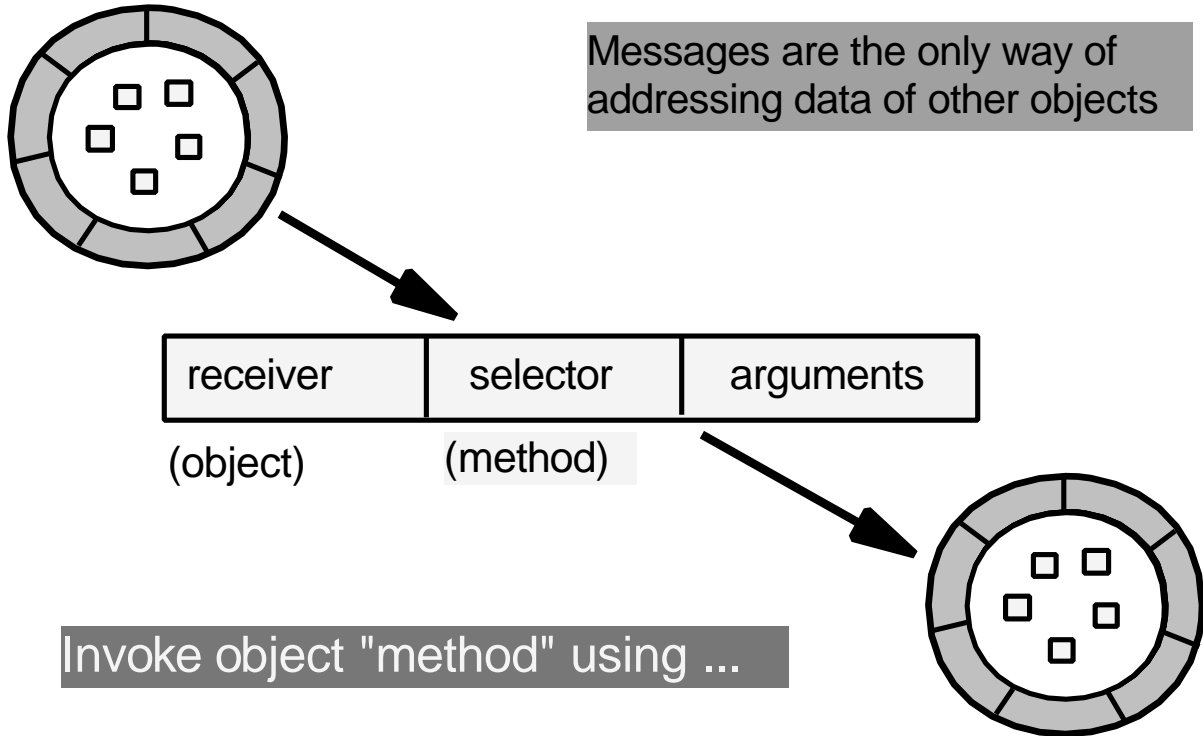
USAGE OBJECT REFERENCE

- Specific object references:

USAGE OBJECT REFERENCE $\left[\begin{array}{l} \text{[FACTORY OF] } class\text{-name} \text{ [ONLY]} \\ \text{[FACTORY OF] } ACTIVE\text{-CLASS} \end{array} \right]$

- without ONLY: Valid for class-name or its sub-classes
- with ONLY: Valid for class-name only
- ACTIVE-CLASS: Valid for class of active object
- Predefined object references:
 - SELF identifies the active object
 - SUPER identifies a superclass of the object in which it is specified
- Usage of object references:
 - in explicit or implicit messages (method invocations)
 - in comparisons on equal/unequal
 - as argument with object or program invocation
 - in assignment operations (SET statement)

Messages



Objects communicate with "messages". Data of another object can only be accessed by sending a message to that object, which causes the execution of a method of the other object.

Such a message has three parts:

1. The *receiver* of the message. This is an object reference that points to the specific object.
2. The *selector*. The selector is the name of the receiver's method to be activated.
3. *Arguments*, to be passed to the selected method, analogous to the arguments in a CALL statement.

Messages in COBOL

- Explicit message through Invoke statement

```

INVOKE object-reference "method" [ USING 

|    |           |
|----|-----------|
| BY | REFERENCE |
| BY | CONTENT   |
| BY | VALUE     |

 {argument ...} ]
[ RETURNING return-item ]
- object reference:
  * class name
  * 'object reference'
- method:
  * Literal
  * Data element (for 'universal ' O-R only)

```

- Implicit message through inline method invocation:

object-reference :: *method* [(*parameter...*)] ...

- Usage in place of identifiers, analogous to intrinsic functions

Creating Objects

COBOL Class BASE

- COBOL provided class BASE, containing the essential methods for creating objects
- Predefined Methods of Base
 - 'New'
 - * class method
 - * creates an object of a given class
 - 'FactoryObject'
 - * object method
 - * gives access to the factory object of the class of a given object
 - ... (TBD)

Creating Objects in COBOL

- Direct or indirect invocation of "New"
- Returns an object reference to the created object
- Example:

```
01 handle-b usage object reference class-b
...
    Invoke class-b "New" returning handle-b
    Invoke handle-b "object-method-2" ...
...
```

Conformance and interfaces

Interfaces

- Interface (protocol, signature)
 - consists of all externally visible characteristics:
 - * method names
 - * parameter names and their properties
 - every object has an interface
 - every class has two interfaces:
 - * factory interface
 - * object interface

Conformance

- Conformance
 - Interface *a* conforms to interface *b*, when
 - * all methods of *b* are also accessible from *a*
 - * the number and characteristics of corresponding parameters match
 - object with interface *a* "is an" object with interface *b*
- Inheritance
 - Interface of sub-class conforms to interface the super-class (e. g. "convertible is a motor vehicle")
 - COBOL: CLASS-ID. ... INHERITS ...

COBOL INTERFACE definition

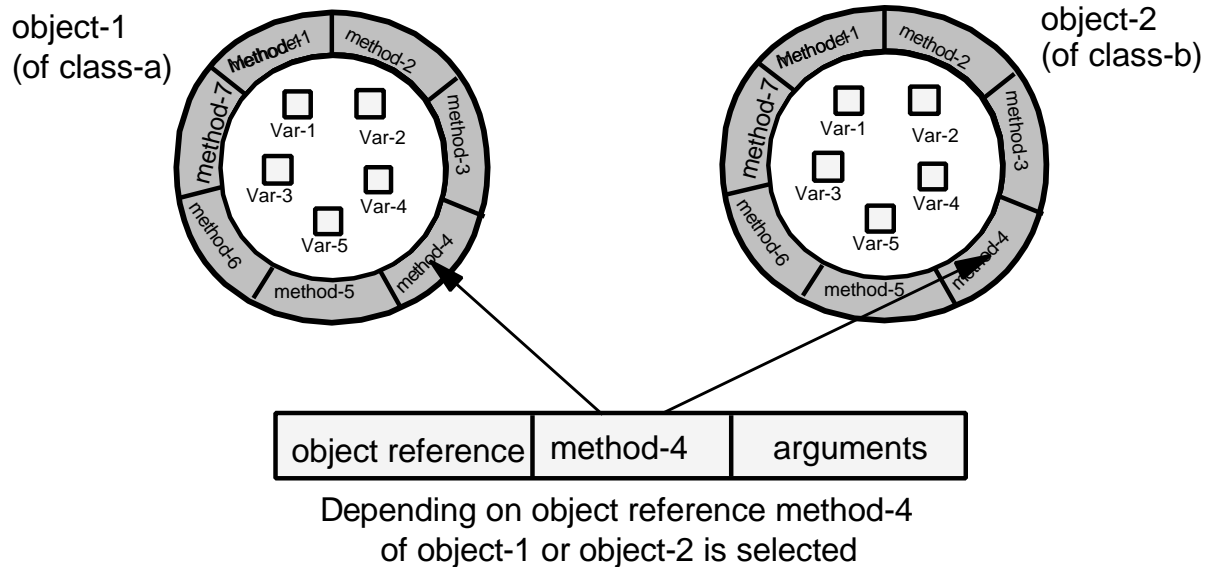
A COBOL interface definition can be used to describe factory or object interfaces independently of a specific class definition. It allows conformance checking even when the class definition is not available. It contains no methods, but "prototypes" only.

```
[ID DIVISION. ]                (parameterized interfaces not covered (see below))
INTERFACE-ID. interface-name-1 [AS literal-1]
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
[{CLASS class-name-1 [AS "external-class-name-1"]}...].
[{INTERFACE int-name-2 [AS "external-int-name-2"]}...].
PROCEDURE DIVISION.
[{method prototypes}]...
END INTERFACE class-name-1.
```

- Interface-specific object reference
OBJECT REFERENCE *interface-name*

- May be used for any objects that implement the interface *interface-name*, e. i. are defined with `IMPLEMENTS interface-name`

Polymorphism



One of the most important characteristics of messages in OOP is that the message itself does not automatically imply its specific effect. This is because a message can be sent to different objects, as long as each of these objects have a method of the specified name that takes the same number and characteristics of arguments, i. e. conforms to the message. There are three cases:

- Hierarchic polymorphism
 - Object reference identifies object in specific class hierarchy
 - COBOL: USAGE OBJECT REFERENCE *class-name* (without 'ONLY')
- Polymorphism using interface definition
 - Object reference identifies an object of an arbitrary class that implements a specific interface
 - COBOL: USAGE OBJECT REFERENCE *interface-name*
- Universal Polymorphism
 - Object reference identifies an object of an arbitrary class
 - COBOL: USAGE OBJECT REFERENCE (without class-name)
 - If the selected object does not conform, an "exception condition" exists (see below)

Other OO Features in COBOL

Object Modifier

- Format

$$\text{object-reference AS } \left. \begin{array}{l} \text{[FACTORY OF] class-name [ONLY]} \\ \text{interface-name} \\ \text{UNIVERSAL} \end{array} \right\}$$

- Allows addressing the object identified by object-reference
 - with the interface of class-name or interface-name, or of the associated factory object
 - as a universal object reference

Property

- Format:

property-1 OF object-identifier-1

- abbreviated way of addressing the value of an object variable

- Prerequisite:

- The object variable property-1 is defined:

$$\text{data-name PROPERTY [WITH NO } \left. \begin{array}{l} \text{GET} \\ \text{SET} \end{array} \right\}]$$

- for use as a sending item:

* without WITH NO GET: implicit generation of method Get Property data-name

- for use as receiving item:

* without WITH NO SET: implicit generation of method Set Property data-name

- Example:

```
Move residence of employee-master to residence of salary-master
                        is equivalent to
Invoke employee-master Get Property residence Returning temp-1
Move temp-1 to temp-2
Invoke salary-master Set Property residence Using temp-2
```

Generic Classes

- Generic class (parameterized class)
- defined by:

```
CLASS-ID. class-name-1 ...
        USING { parm-name-1 }... .
```

- class-name-1 references classes that are fully specified only by the client
- parm-name-1 is a place holder for a class-name to supplied later
- specification in repository of client:

```
class-name-1 ... EXPANDS class-name-2 USING {class-name-3}...
```

- class-name-1 is "instance" of generic class class-name-2
- each class-name-3 replaces the corresponding parm-name-1

Exception Handling

- Performed by "Generalized Exception Handling":

```
>>TURN exception-name CHECKING {ON [WITH LOCATION]}  
                                {OFF}
```

Use after Exception { class-name | interface-name }

Raise EXCEPTION *exception-name*

- intrinsic functions
 - Exception-Status, Exception-Statement, Exception-Location, Exception-Object
 - * OO Exception-names:

```
EC-OO  
EC-OO-CONFORMANCE  
EC-OO-EXCEPTION  
EC-OO-INTRINSIC  
EC-OO-METHOD  
EC-OO-NULL  
EC-OO-RESOURCE  
EC-OO-UNIVERSAL  
EC-OO-IMP
```

- Example

```
...  
Declaratives.  
  Use after exception EC-OO-Exception.  
  ...  
  Display function exception-location function exception-statement  
  ...  
End Declaratives.
```

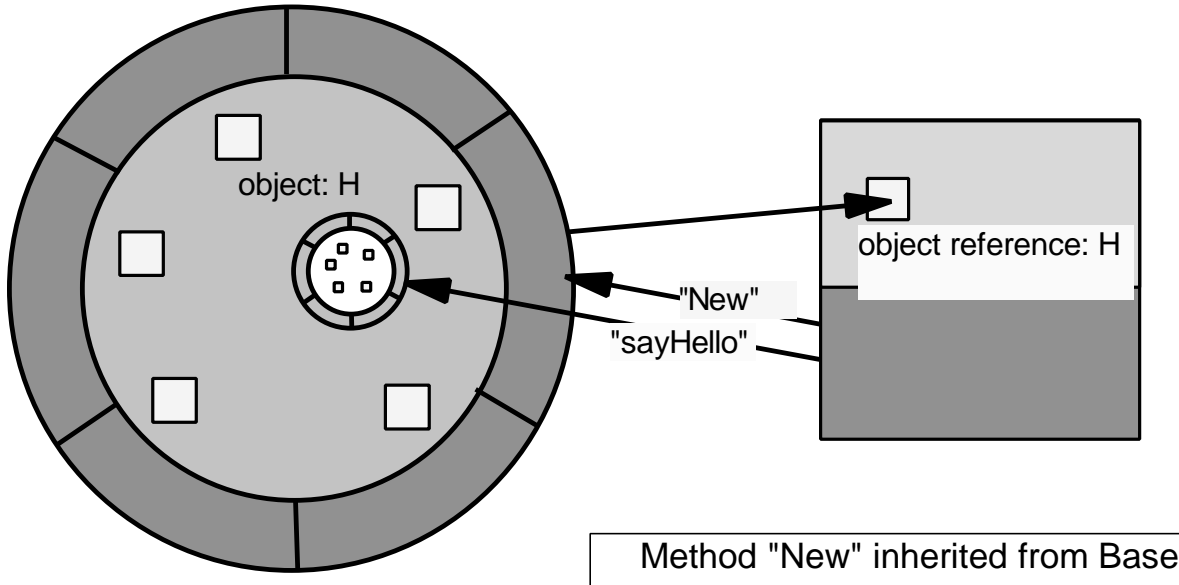
```
...  
>>Turn EC-OO checking on  
...  
>>Turn EC-OO checking off  
...
```

Examples

Example 1

Class: Hello

Program: Client



```

Identification Division.
Class-Id. Hello
    inherits FjBase.
Environment Division.
Configuration Section.
Repository.
    Class FjBase.
    
```

```

Identification Division.
Object.
Procedure Division.
    
```

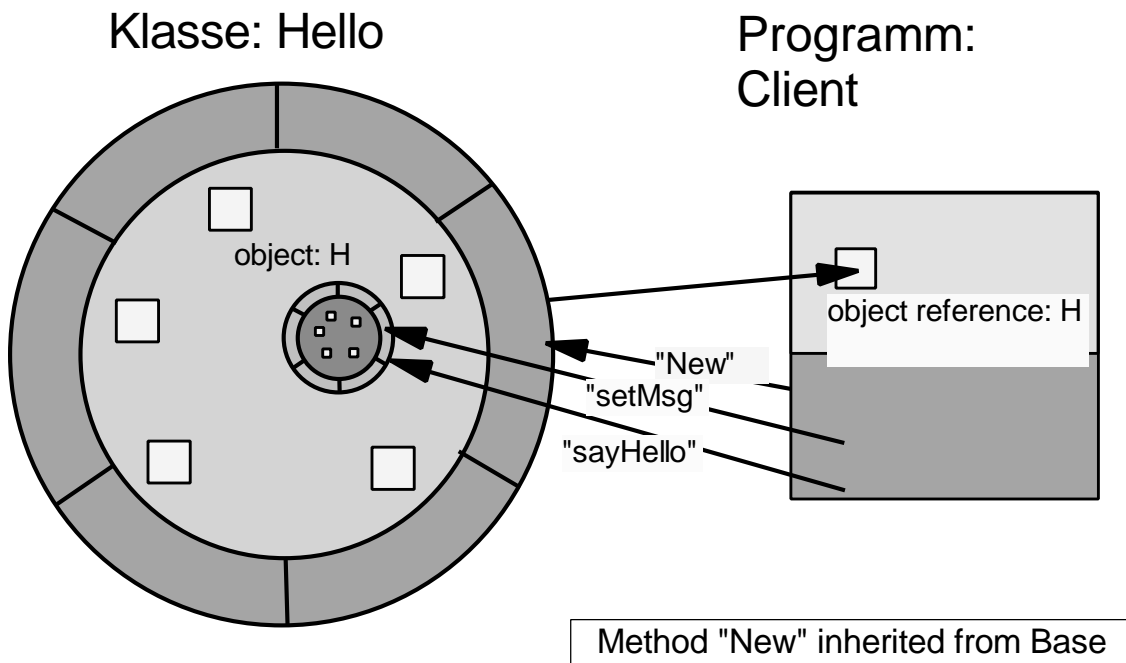
```

Identification Division.
Method-Id. sayHello.
Procedure Division.
    Display "Hello World!".
End Method sayHello.
End Object.
End Class Hello.
    
```

```

Identification Division.
Program-Id. Client.
Environment Division.
Configuration Section.
Repository.
    Class Hello.
Data Division.
Working-Storage Section.
01 H object reference hello.
Procedure Division.
    Invoke Hello "new" returning H
    Invoke H "sayHello"
    Exit Program.
End Program Client.
    
```

Example 2 (with attribute)



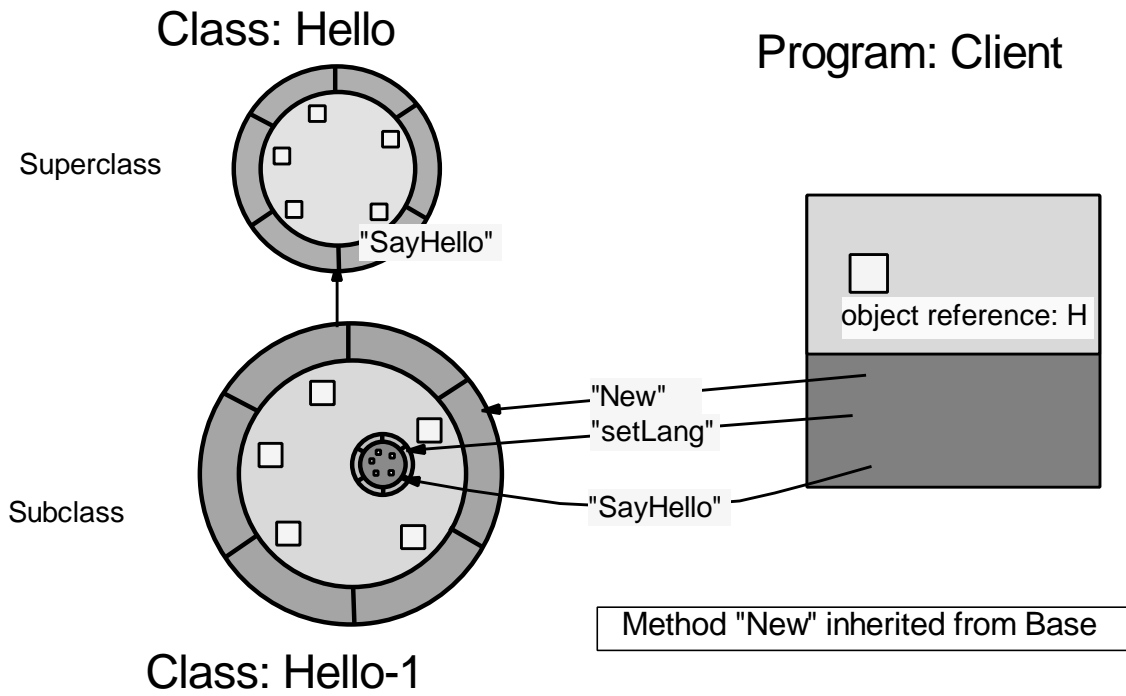
```
Identification Division.  
Class-Id. Hello inherits FjBase.  
Environment Division.  
Configuration Section.  
Repository.  
    Class FjBase.  
Identification Division.  
Object.  
Data Division.  
Working-Storage Section.  
01 msg pic x(30).  
Procedure Division.
```

```
Identification Division.  
Method-Id. sayHello.  
Procedure Division.  
    Display msg.  
End Method sayHello.
```

```
Identification Division.  
Method-Id. setMsg.  
Data Division.  
Linkage Section.  
01 in-msg pic x(30).  
Procedure Division using in-msg.  
    Move in-msg to msg.  
End Method setMsg.  
End Object.  
End Class Hello.
```

```
Identification Division.  
Program-Id. Client.  
Environment Division.  
Configuration Section.  
Repository.  
    Class Hello.  
Data Division.  
Working-Storage Section.  
01 H usage object reference hello.  
Procedure Division.  
    Invoke Hello "new" returning H  
    Invoke H "setMsg" using by content "Hello World!"  
    Invoke H "sayHello"  
    Exit Program.  
End Program Client.
```

Example 3 (with inheritance)



```

Identification Division.
Class-Id. Hello1 inherits Hello.
Environment Division.
Configuration Section.
Repository.
    Class Hello.

```

```

Identification Division.
Object.
Data Division.
Working-Storage Section.
01 lang-table.
    02 pic x(31) value "EHello World from Sample 3!".
    02 pic x(31) value "DHallo Welt von Beispiel 3!".
    02 pic x(31) value "FAttention monde d'exemple 3!".
01 lang-table-x redefines lang-table.
    02 lang occurs 3 indexed by x1.
        03 lang-code pic x.
        03 lang-msg pic x(30).

```

```

Procedure Division.

```

```

Identification Division.
Method-Id setLang.
Data Division.
Linkage Section.
01 in-lang pic x.
Procedure Division using in-lang.
    Set x1 to 1
    Search lang
        At End
            Invoke self "setMsg"
                using by content "Wrong language!"
            When lang-code(x1) = in-lang
                Invoke self "setMsg"
                    using by content lang-msg(x1)
        End-Search
    Exit Method.
End Method setLang.
End Object.
End Class Hello1.

```

```

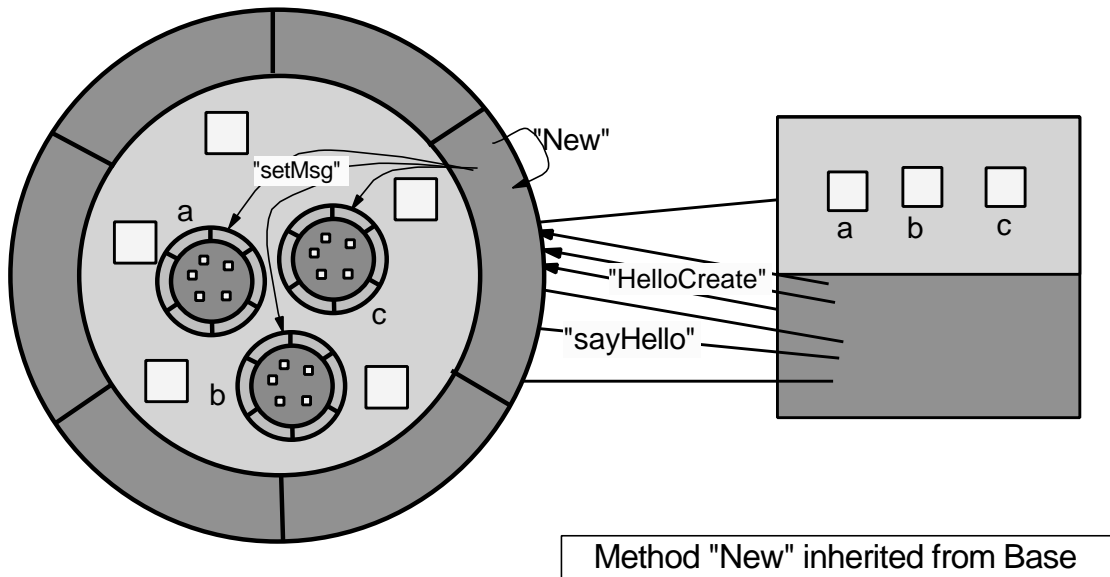
Identification Division.
Program-id. Client.
Environment Division.
Configuration Section.
Repository.
    Class Hello1.
Data Division.
Working-Storage Section.
01 H usage object reference Hello1.
Procedure Division.
    Invoke Hello1 "New" returning H
    Invoke H "setLang" using "F"
    Invoke H "sayHello"
    Exit Program.
End Program Client.

```

Example 4 (with factory)

Class: Hello

Program: Client



```

Identification Division.
Class-Id. Hello inherits FjBase.
Environment Division.
Configuration Section.
Repository.
    Class FjBase.

Identification Division.
Factory.
Procedure Division.

Identification Division.
Method-Id. HelloCreate.
Data Division.
Linkage Section.
01  in-msg pic x(30).
01  anObject usage object reference Self.
Procedure Division using in-msg returning anObject.
    Invoke self  "New" returning anObject
    Invoke anObject "setMsg" using in-msg.
End Method HelloCreate.

End Factory.

Identification Division.
Object.
Data Division.
Working-Storage Section.
01  msg pic x(30).
Procedure Division.

Identification Division.
Method-Id. sayHello.
Procedure Division.
    Display msg.
End Method sayHello.

Identification Division.
Method-Id. setMsg.
Data Division.
Linkage Section.
01  in-msg pic x(30).
Procedure Division using in-msg.
    Move in-msg to msg.
End Method setMsg.

End Object.

End Class Hello.

```

```
Identification Division.
Program-Id. Client.
Environment Division.
Configuration Section.
Repository.
    Class Hello.
Data Division.
Working-Storage Section.
01 a usage object reference hello.
01 b usage object reference hello.
01 c usage object reference hello.
Procedure Division.
    Invoke Hello "HelloCreate"
        using by content "Hello from A" returning a
    Invoke Hello "HelloCreate"
        using by content "Hello from B" returning b
    Invoke Hello "HelloCreate"
        using by content "Hello from C" returning c
    Invoke a "sayHello"
    Invoke b "sayHello"
    Invoke c "sayHello"
    Exit Program.
End Program Client.
```