

**Chapter
1**

**GETTING
STARTED**

*Get on the
Fast Track!*



TM

**SYS-ED/
Computer
Education
Techniques, Inc.**

Objectives

You will learn:

- Features and advantages of Micro Focus Net Express.
- Net Express components.
- Object oriented programming concepts.
- Objects in programs.
- Class declaration.
- Object references.
- Creating and destroying objects.
- Instance creation methods.
- Get and set property methods.
- Class libraries.



1 What is Net Express?

Net Express is:

- An integrated collection of tools used for creating and extending enterprise business processes written in COBOL which deliver distributed e-business COBOL applications on Windows or UNIX.
- Used for simplifying the editing, compiling, and debugging when managing projects.
- An editor tailored to the needs of COBOL programmers for modifying code.
- Designed to leverage existing applications, data and programming resources.
- Used for constructing distributed applications across a Windows, LINUX or UNIX environment which can interface with e-business systems written in Java or under the Microsoft .NET environment.

Net Express Wizards:

- Can be used to build original or mined components as Enterprise JavaBeans (EJBs) or COM objects.
- Generate native code that can be optimized for peak performance on specific hardware platforms.
- Create extensive COBOL dialect support.
- Create profiling reports.

1.1 Facilities

OpenESQL	Translates embedded SQL statements into ODBC API calls; this allows applications to be developed which access different database systems. Any data source for which an ODBC driver is available can be used.
OpenESQL Assistant	Provide programmer's with the capability to build SQL statements through a point-and-click interface and integrated data tools.
Native access support to DB2	Integrated support for pre-compilers from Oracle and Sybase.
SCP: Server Control Program	Server software for use on UNIX. It must be installed, in order to use the UNIX Option of Net Express.
XML-enabled COBOL	Set of XML syntax extensions that can be coded in the COBOL program. The syntax extensions enable the COBOL program to perform input and output on an XML instance document rather than using a traditional file method such as an indexed file.

2 Major Components

Component	Explanation
IDE: Integrated Development Environment	Windows systems, with pull-down menus, in which you edit, compile and debug your code, and via which you access most of the Net Express tools.
Project	File detailing all the files in an application, and how they should be compiled and linked. A project should be created for every application.
Data Tools	Set of tools which provides the capability to convert, browse, edit and create data files used by an application.
HTML Page Wizard	Tool used for creating the user interfaces for web applications. Files can be in any COBOL format and be viewed at both the record and field levels.
Internet Application Wizard	Tool used to create web applications.
Form Designer	WYSIWYG form editor.
Solo	Web server software.
Windows GUI Application Wizard	Tools used for creating Windows-based applications.
Dialog System	Tool used to design windows and dialog boxes for Windows-based applications. Procedures can also be written for handling events.

3 Object Oriented COBOL Programming

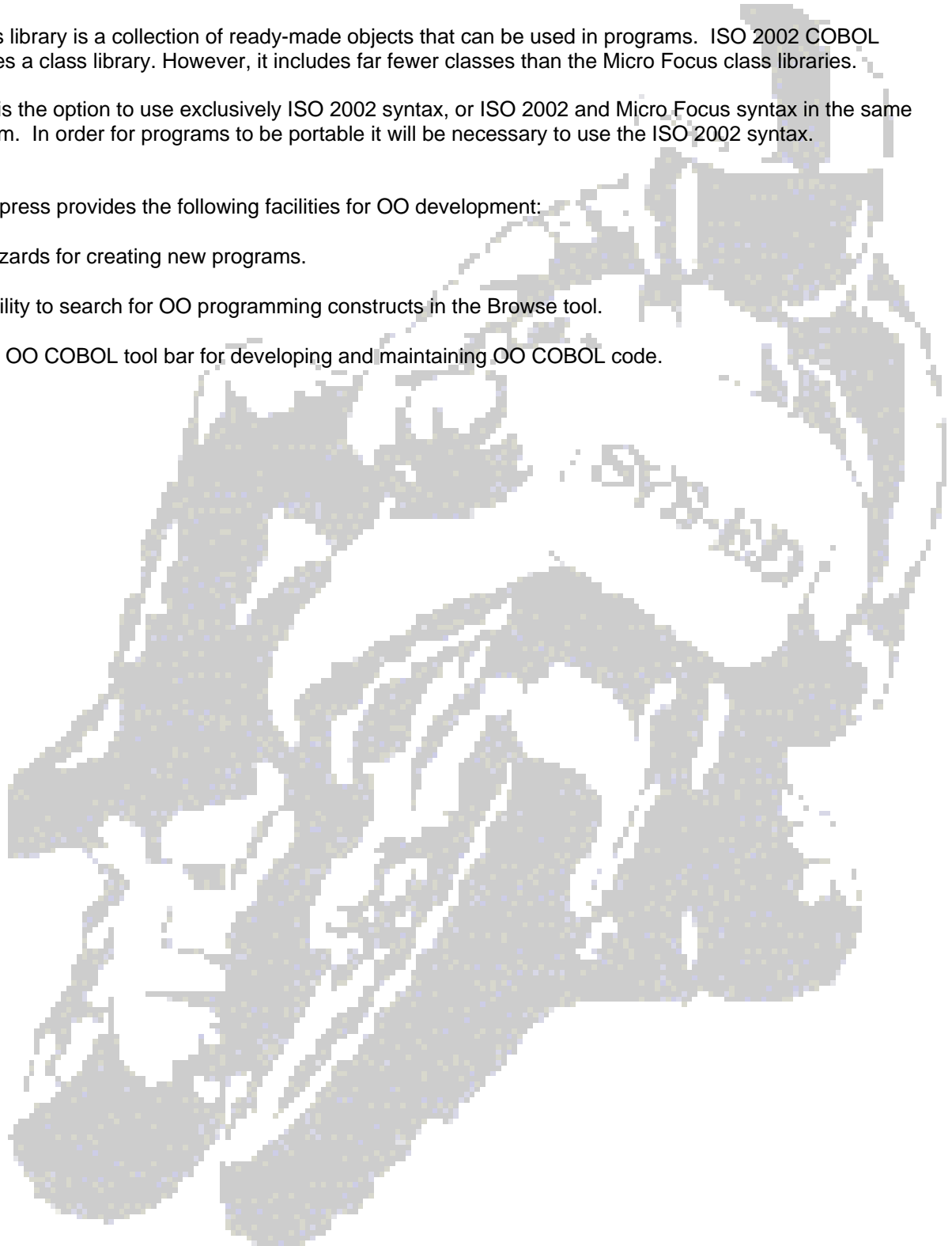
The Micro Focus Net Express system has the capability to develop object-oriented (OO) programming in COBOL.

A class library is a collection of ready-made objects that can be used in programs. ISO 2002 COBOL provides a class library. However, it includes far fewer classes than the Micro Focus class libraries.

There is the option to use exclusively ISO 2002 syntax, or ISO 2002 and Micro Focus syntax in the same program. In order for programs to be portable it will be necessary to use the ISO 2002 syntax.

Net Express provides the following facilities for OO development:

- Wizards for creating new programs.
- Ability to search for OO programming constructs in the Browse tool.
- An OO COBOL tool bar for developing and maintaining OO COBOL code.



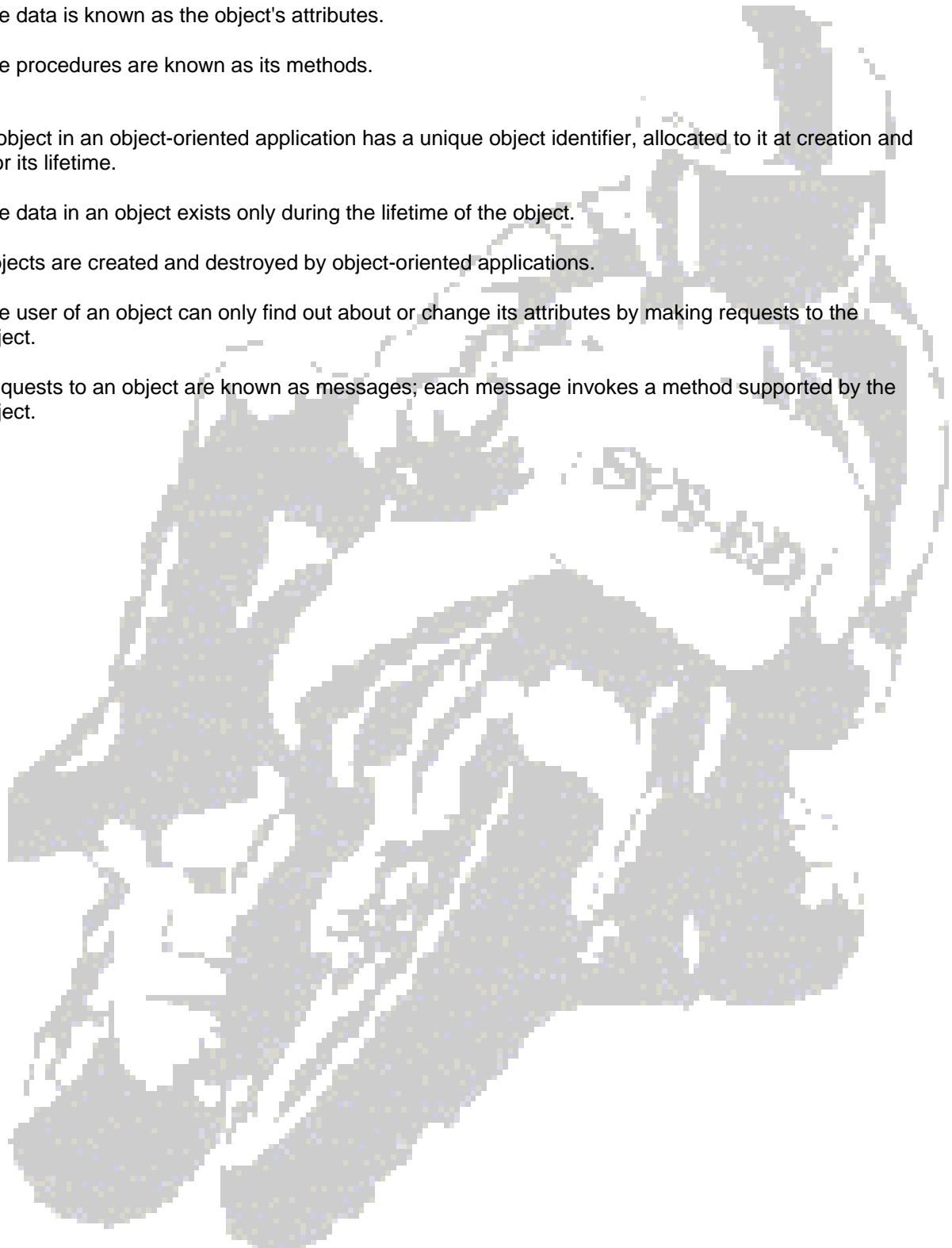
4 Objects

An object is a combination of data and the procedures to operate on that data.

- The data is known as the object's attributes.
- The procedures are known as its methods.

Every object in an object-oriented application has a unique object identifier, allocated to it at creation and fixed for its lifetime.

- The data in an object exists only during the lifetime of the object.
- Objects are created and destroyed by object-oriented applications.
- The user of an object can only find out about or change its attributes by making requests to the object.
- Requests to an object are known as messages; each message invokes a method supported by the object.

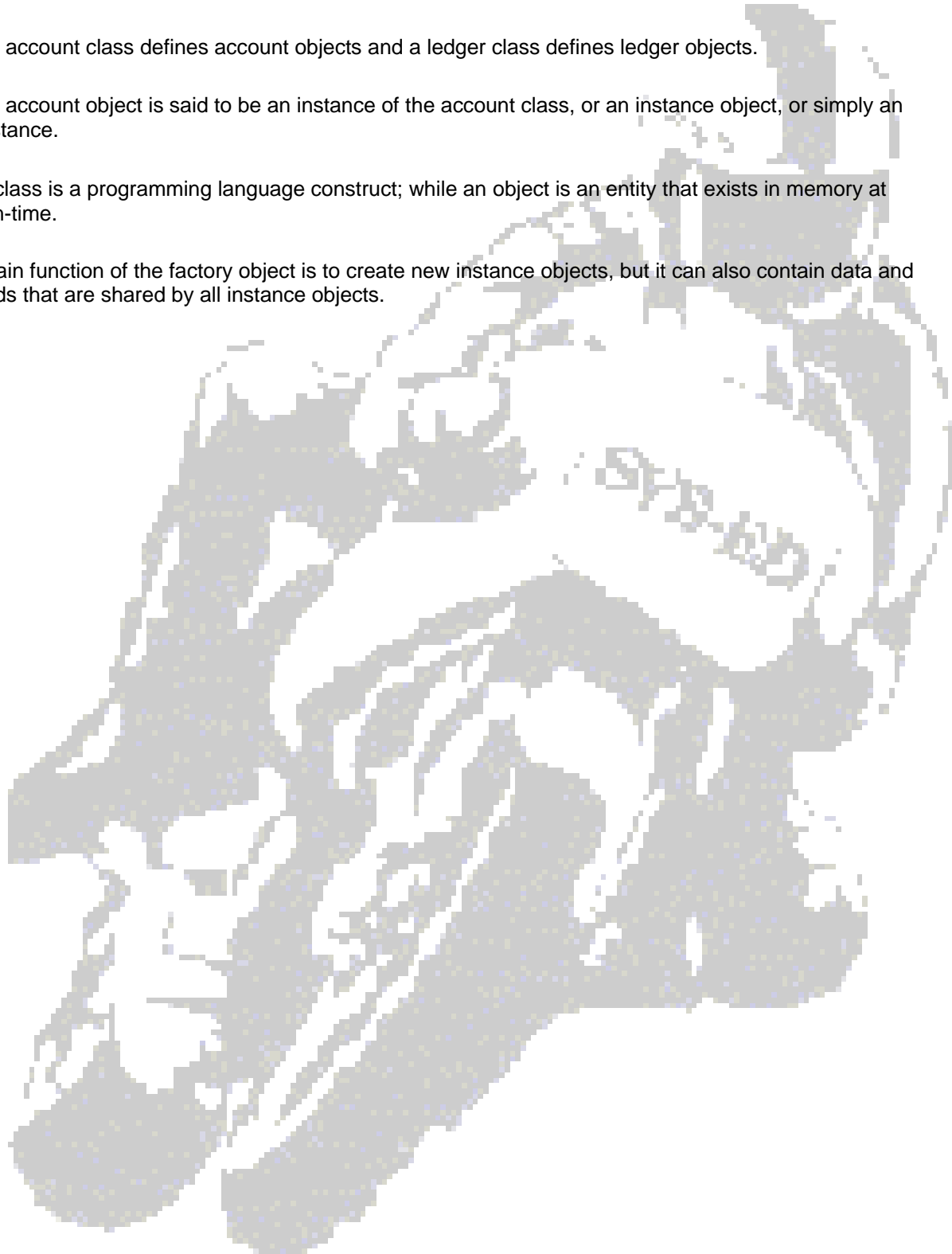


5 Classes

A class is a definition of an object; it embodies all the information needed to create and manipulate objects of a particular type.

- An account class defines account objects and a ledger class defines ledger objects.
- An account object is said to be an instance of the account class, or an instance object, or simply an instance.
- A class is a programming language construct; while an object is an entity that exists in memory at run-time.

The main function of the factory object is to create new instance objects, but it can also contain data and methods that are shared by all instance objects.



6 Methods

Methods are the pieces of code that implement the behavior of an object.

In OO COBOL, each method is a separate source element nested within the factory or object source element.

- An object method can access its own data, the instance data and the factory data declared in the factory object source element.
- A factory method can access its own data and the factory data.

6.1 Method Prototype

Methods can be incomplete; these are known as method prototypes.

A method prototype does not contain any code, just a heading and an end-marker. A method prototype is always fully implemented elsewhere in the application.

7 Interfaces

Interfaces are collections of method prototypes. The set of method prototypes defines a common behavior that a variety of objects might share.

Interfaces provide additional flexibility in designing OO applications; they are one of the elements in an OO language that provide polymorphism.



8 Messages

A message is the way a request is made to an object for performing a service.

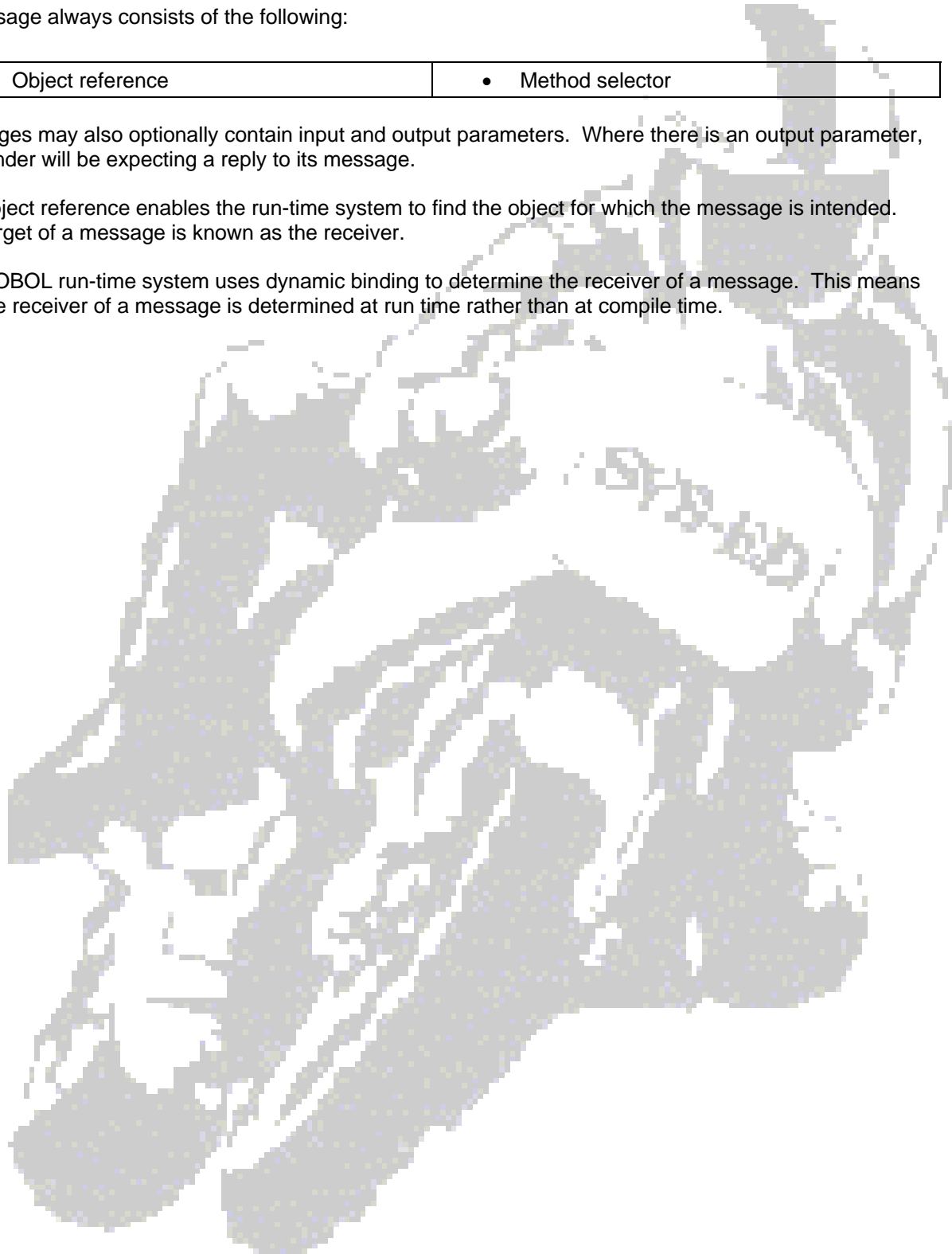
A message always consists of the following:

• Object reference	• Method selector
--------------------	-------------------

Messages may also optionally contain input and output parameters. Where there is an output parameter, the sender will be expecting a reply to its message.

The object reference enables the run-time system to find the object for which the message is intended. The target of a message is known as the receiver.

The COBOL run-time system uses dynamic binding to determine the receiver of a message. This means that the receiver of a message is determined at run time rather than at compile time.



9 OO Topics

Encapsulation

Encapsulation in OO languages is the inclusion within an object of everything it needs to function: its data and the implementation of its methods.

Other objects can use it as long as they adhere to the object's interface; they do not need to know anything about the way the object holds and manipulates its data.

Encapsulation helps to preserve the integrity of data.

Inheritance

Inheritance enables OO languages to mirror the real world more closely by establishing hierarchies of classes.

Most objects in the real world belong to specific categories; for example, cars and motor bikes are vehicles, managers and workers are employees.

Using inheritance in OO applications reduces coding effort, because methods only need to be included in a subclass if the required behavior is different from that provided in the superclass.

When creating a method in a subclass of the same name as a method in the class it inherits from, the method in the subclass overrides the method in the class. The method in the subclass is known as a reimplementation of the method in the class.

Polymorphism

Polymorphism is an important part of any object-oriented programming language. Polymorphism means that the same message sent to different objects can invoke different methods.

Polymorphism is provided by two mechanisms in OO COBOL:

- Where a number of subclasses inherit from a class, the methods in the class can be method prototypes, which are methods that do not contain any code. Complete methods of the same name appear in each subclass, and these can be different.
- Interfaces are collections of methods applicable to a variety of objects. The methods in the interface are prototypes. If a class implements an interface, it must provide complete methods of the same name, and these methods can differ between classes that implement the same interface.

10 Objects in Programs

Programs that use objects all have certain features in common:

They have a Repository paragraph.	It lists all the classes and interfaces the program is going to use. If the program itself is an OO COBOL class, the Repository paragraph also lists its superclass.
They declare one or more data items of type OBJECT REFERENCE.	An object reference data item holds an object handle, which is a pointer to the location of the object in memory. The run-time system assigns a unique object handle to every single object active in an application. An object handle enables messages to be sent to the object.
They send messages.	When sending a message to an object, a method is invoked inside the object. Some methods receive or return parameters; when a method it will be necessary to include the parameters as part of the message.

11 Class Declaration

In order to use an object in a program, the object's class must be declared in the program's Repository paragraph.

The CLASS clause also:

- Creates a data item for each class named. At run time, this data item holds an object handle to the factory object.
- Enables the run-time system to find and load the file with the class executable code.

When a class is declared, it effectively makes the class's factory object available. The class names in the Repository paragraph are automatically declared as object reference data items. This enables messages to be sent to the factory object.

This code shows class registration for classes CharacterArray and Employee:

```
repository.  
    class CharacterArray as "chararry"  
    class Employee
```

When an AS clause is coded, the Compiler uses the value of the literal as the external name of the class program. Otherwise; it uses the class name changed to upper case.

12 Object References

An object reference can hold a handle to a factory or instance object. Depending on the syntax being used, object references are untyped or typed.

- An untyped object reference data item can hold an object reference for any factory or instance object.
- Untyped object references are also known as universal object references. A typed object reference can only hold an object reference of the type specified by the syntax.

It is a good programming practice to use typed object references rather than untyped. This is because it allows the Compiler to perform its conformance checking, which tends to lessen the number of run-time errors that occur.

12.1 Object References: Declaring

It will be necessary to declare data items of type OBJECT REFERENCE in order for holding handles to any objects that will be used.

01 anObject	usage object reference.
01 secdObject	usage object reference factory of BankAccount.
01 thirdObject	usage object reference active-class.
01 fourthObject	usage object reference Rentable.

13 Object References: Manipulating

Only certain direct operations can be used on object references.

Send a message to the object represented by the handle in the object reference.

Example:

```
invoke anObject "message"
```

Copy an object reference to another.

Example:

```
set anObject1 to anObject2
```

Test whether two object references refer to the same object.

Example:

```
if anObject1 = anObject2 ...
```

14 Messages: Sending

Sending a message to an object will make it execute a method. The method to be executed is named in the message. When an object does not understand a message, it is passed up the method inheritance chain until it is recognized and executed.

There are three ways of sending a message:

- | | | |
|--------------------|-----------------------------|---------------------|
| • INVOKE statement | • In-line method invocation | • Object properties |
|--------------------|-----------------------------|---------------------|

A message can be sent by using the INVOKE statement.

Example:

```
invoke myClass "new" returning myObject
```

This sends message "new" to myClass; myClass is an object reference referring to a previously declared class.

The target of an INVOKE statement is always an object reference. USING and RETURNING parameters are optional, and follow the same rules as a COBOL CALL statement.

A literal does not have to be used for the message name. A message name can also be put into a PIC x(n) data item.

Example:

```
move "new " to aMessage  
...  
invoke myClass aMessage returning myObject
```

15 In-line Method Invocation

A method can be invoked in-line, as long as it has a returning parameter, by coding the object reference followed by the method which is separated by two colons in place of an operand in a statement.

Example:

```
move anAccount:"getBalance" to current-balance
```

This is equivalent to the following INVOKE statement:

```
invoke anAccount "getBalance" returning current-balance
```

15.1 Invocation Using Object Properties

Object properties can be used to invoke get property and set property methods. Get and set property methods are either generated automatically by the Compiler in response to object property syntax in data definitions, or coded by the programmer using special syntax in the Method-ID paragraph.

In order to invoke a get or set property method, it will be necessary to use a statement such as:

```
move balance of anAccount to current-balance
```

This statement invokes the get property method balance of the account object referred to by the object reference anAccount.

16 New Instance Object: Creation

When creating a new object, the run-time system allocates an object handle, and storage for the variables declared in the instance object's Working-Storage Section.

A programmer can create as many instances of any particular class of object as system resources allow. Each instance has identical behavior, but its own unique data. Classes either provide their own methods for creating new object instances, or inherit them from a superclass.

The method "new" is implemented in the class Base, which is part of the Base Class Library; it is used to create an instance of a single object. In order to create an instance object, it will be necessary to send the message to a class and provide a variable to receive the object reference.

Example:

```
working-storage section.  
01 anAccount      object reference of BankAccount.  
...  
procedure division.  
...  
    invoke BankAccount "new" returning anAccount
```

17 Destroying Objects

Some OO programming languages perform garbage collection at run time; they automatically destroy objects that are no longer in use. OO COBOL does not provide garbage collection. Once an object has been created, it remains in existence until destroyed explicitly. This is the case, even if the data item which holds its object handle is destroyed or goes out of scope.

There are two ways an object can be destroyed:

- The application that created it can destroy it when it has finished with it.
- The run-time system always destroys any objects still in existence when the run unit terminates.

It is good programming practice to destroy unwanted objects in applications, rather than leaving it to the run-time system. An object should not be destroyed if it is still in use.

In order to destroy an object, send it the "finalize" message.

Example:

```
invoke anObject "finalize" returning anObject
```

18 Classes

It will be necessary to write OO COBOL classes for creating objects other than the ones in the class libraries supplied by Micro Focus.

There are three entities which need to be recognized when writing a class.

Class	The source code defining the class.
Factory object	The class at run time.
Instance objects	Created by the factory object at run time.

A class embodies all the behavior for its factory object and its instance objects. It defines the factory and instance data, and factory and instance methods. A class also inherits the data and methods of its superclass.

Each class describes the behavior of two different objects:

<ul style="list-style-type: none">• The factory object.	<ul style="list-style-type: none">• The instances of the class.
---	---

There is never more than one occurrence of each type of factory object in an application, whereas there can be many occurrences of its instances.

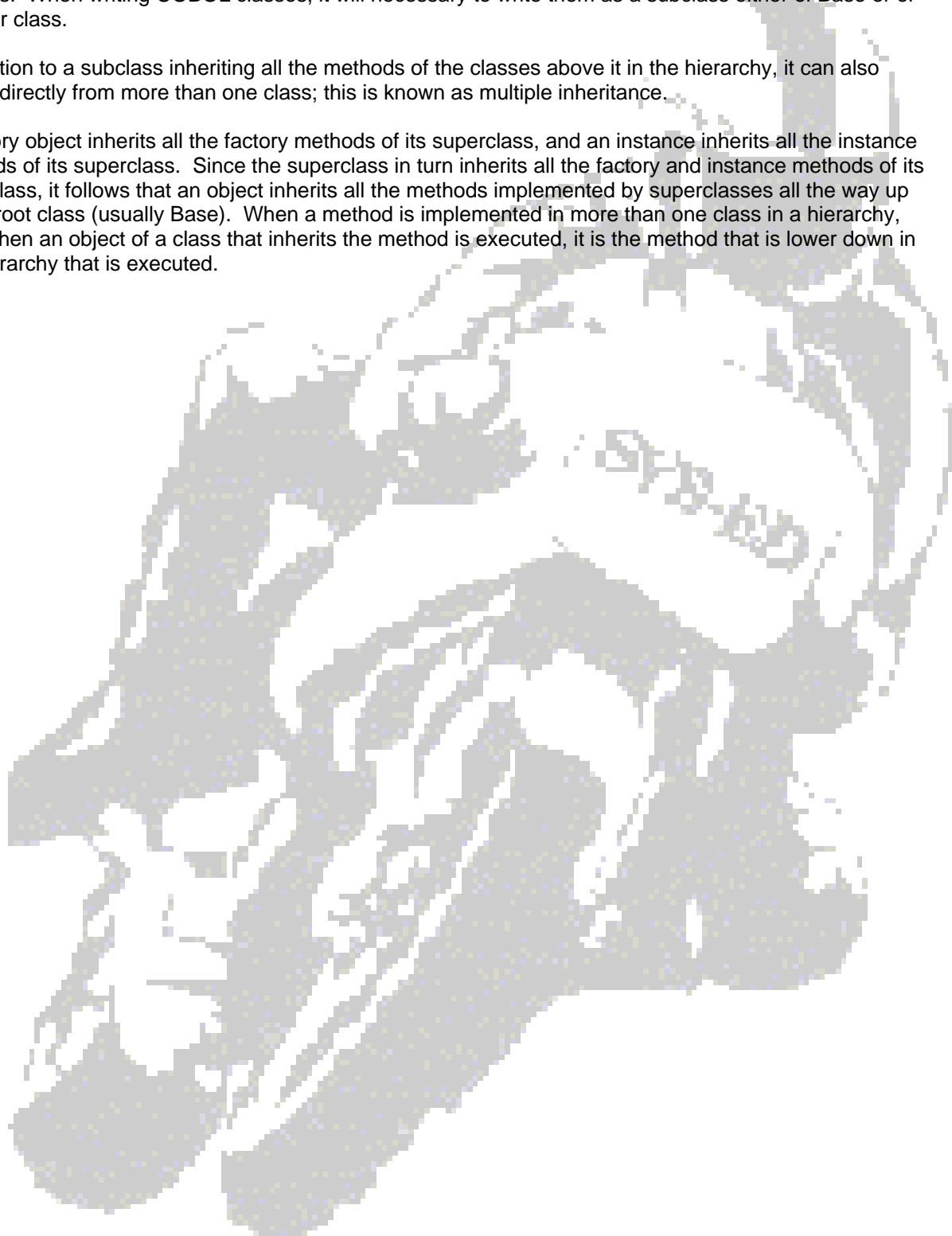
The main function of any factory object is to create instances, although in some cases a factory object has other behavior as well.

19 Inheritance

Most OO COBOL classes inherit from a superclass. In the Micro Focus class libraries, the only exception to this is the Base class, which is at the root of the inheritance tree for all other classes in the class libraries. When writing COBOL classes, it will necessary to write them as a subclass either of Base or of another class.

In addition to a subclass inheriting all the methods of the classes above it in the hierarchy, it can also inherit directly from more than one class; this is known as multiple inheritance.

A factory object inherits all the factory methods of its superclass, and an instance inherits all the instance methods of its superclass. Since the superclass in turn inherits all the factory and instance methods of its superclass, it follows that an object inherits all the methods implemented by superclasses all the way up to the root class (usually Base). When a method is implemented in more than one class in a hierarchy, then when an object of a class that inherits the method is executed, it is the method that is lower down in the hierarchy that is executed.



20 Methods

Methods are written inside class source elements. Each method is a nested source element bracketed by METHOD-ID and END METHOD headers. It is possible to optionally precede the first METHOD-ID header by an IDENTIFICATION DIVISION header.

Factory methods can access factory Working-Storage data and instance methods can access instance Working-Storage data. In all other respects factory and instance methods look and work identically.

A method can also have data of its own; or a programmer can declare the following types of storage section in a method:

Local storage	Data items that are local to the current invocation of a method.
Linkage	Data items for passing parameters to and from a method.

```

method-id. setInItemA.
local-storage section.
01 aTempltem          pic x(4) comp-5.
linkage section.
01 InkValue          pic x.
01 aResult           pic x comp-5.
procedure division using InkValue
    returning aResult.
    move InkValue to inItemA
    move 1 to aTempltem
    call "CBL_TOUPPER" using inItemA
        by value aTempltem
    if InkValue = cItemA
        move 1 to aResult
    else
        move 0 to aResult
    exit method.
end method setInItemA.

```

21 Predefined Object Reference Names

Three predefined object reference names, which are initialized by the run-time system when an object is created, are available to class source elements to enable them to send messages to themselves.

Object Reference Name	Description
SELF	<p data-bbox="488 464 1341 520">Enables an object to send a message to itself. The method invoked is a method in the same class.</p> <ul data-bbox="488 583 1403 716" style="list-style-type: none"><li data-bbox="488 583 1403 640">• If a message is sent to SELF from object A, object A is the receiver of the message.<li data-bbox="488 657 1403 716">• If a message is sent to SELF from object C, object C is the receiver of the message.
SUPER	<p data-bbox="488 741 1341 798">Enables an object to send a message to itself. The method invoked is a method in one of the superclasses of the class.</p> <ul data-bbox="488 840 1403 961" style="list-style-type: none"><li data-bbox="488 840 1403 961">• If SUPER is used from an instance, the run-time system searches for a method beginning with the instance code of the superclass immediately above the class, and works its way up through the instance methods of all the superclasses until it finds a method matching the message.
SELFCLASS	<p data-bbox="488 978 1393 1066">Micro Focus extension that enables an instance object to send a message to the factory object which created it. The method invoked is a method in the factory object.</p> <ul data-bbox="488 1108 1419 1165" style="list-style-type: none"><li data-bbox="488 1108 1419 1165">• If SELFCLASS is used from a factory method, it points to the metaclass for this class, which is an instance of Behavior.

22 Instance Creation Methods

It will be necessary to create instance objects from classes. At the most basic level, creating a new object instance requires a memory allocation for its data and an object handle by which it can be referred to. This task is handled by an instance method in the metaclass Behavior, called "new". All factory objects are instances of the metaclass, and inherit this method.

If the decision has been made not to carry out any initialization for an instance, it will not be necessary to code anything in the class for its creation. The "new" method can be inherited from the metaclass.

Example:

```
class-id. simple inherits from base.  
* No factory methods for simple  
...  
object.  
working-storage section.  
    01 simpleInstanceData          pic x(80).  
* Instance methods for simple are not shown here.  
...  
end object.  
end class simple.
```

This code would provide an application with a new instance of Simple:

```
invoke simple "new" returning anObject
```

23 Get and Set Property Methods

A common coding requirement will be to set a number of data fields to particular values. In order to meet this requirement in an OO application, a method will be needed for each field that sets the required value. Methods that set values are known as set methods.

Another frequent coding requirement is the retrieval of information from a number of data fields. In order to meet this requirement in an OO application, a method will be needed for each field that gets the required value. Methods that get values are known as get methods.



24 PROPERTY Clause in a Data Item Definition

If the PROPERTY clause is coded in a data item definition, the Compiler will automatically generate the simple get property and set property methods. These methods differ slightly depending on the data description.

Example:

```
class-id. A inherits from Base.  
...  
factory.  
...  
end factory.  
object.  
  working-storage section.  
    01 custNo pic 9(10) comp property.  
  ...  
end-object.  
end class A.
```

The compiler generates get property and set property methods for cust-no, with internal method names. It is not necessary to know what these method names are, because they are not called directly.

25 PROPERTY Clause in a Method Definition

With this mechanism, get and set methods are written, but use the PROPERTY clause in the Method-ID paragraph of each method. The compiler will then generate internal method names which will work with the object property syntax.

This mechanism is used when methods are needed which do more than the simple methods generated when the PROPERTY clause is specified in a data definition.

Example:

```
class-id A inherits from Base.  
...  
object.  
working-storage section.  
01 custNo pic 9(10).  
method-id. get property custNo.  
linkage section.  
01 lkCustNo pic 9(10).  
procedure division returning lkCustNo.  
  *> My other code here  
  move custNo to lkCustNo  
  *> My other code here  
  exit method.  
end method.  
  
method-id. set property custNo.  
linkage section.  
1 lkCustNo pic 9(10).  
procedure division using lkCustNo.  
  *> My other code here  
  move lkCust-no to custNo  
  *> My other code here  
  exit method.  
end method.
```

26 Interfaces

An interface is a set of method prototypes; which are skeleton methods with no code. Classes can implement interfaces. A class that implements an interface must provide full method definitions for the method prototypes in the interface.

A programmer can develop a hierarchy of inheriting interfaces, similar to a hierarchy of classes. An interface can inherit from more than one interface; this is known as multiple inheritance. An inheriting interface has all the method specifications defined for the inherited interface definition or definitions, including any method specifications that the inherited definition or definitions inherited.

The inheriting interface can define new methods augmenting the set of inherited method specifications. The inheriting interface must always conform to each of the inherited interfaces.

26.1 Interface Implementation

Factory objects and instance objects within classes can implement interfaces. The implementing object implements all the method prototypes defined for the implemented interface definition or definitions, including any method prototypes that the implemented definition or definitions inherited.

Example:

This instance object implements the Drivable interface.

class-id. Car inherits from Vehicle.

configuration section.

repository.

```
class Car
class Vehicle
interface Drivable
```

.

factory.

* The code for the factory object starts here

...

* The code for the factory object ends here.

end factory.

*-----The instance object code starts here.

object. implements Drivable.

working-storage section. *> Instance data goes here.

...

procedure division.

interface-methods. *> Start of implementation

*> of interface methods.

method-id. "calcMileage".

data division.

linkage section.

01 endMileage pic 9(6).

01 beginMileage pic 9(6).

01 mileage pic 9(6).

procedure division using endMileage, beginMileage

returning mileage.

subtract endMileage from beginMileage giving mileage.

end method "calcMileage".

method-id. "logDamage".

...

end method "logDamage".

method-id. "pickUp" *> Implementation of method

... *> inherited by Drivable from

end method "pickUp" *> Rentable

instance methods. *> start of instance methods

*> for an object of this class

...

end object.

*-----the instance object code ends here.

end class Car.

27 Class Libraries

The following class libraries are supplied:

Class Library	Explanation
Base	The Base Class Library provides COBOL classes for basic system support. It also provides classes for <ul style="list-style-type: none">• Handling exceptions raised by objects.• Treating COBOL intrinsic data types as objects.• Managing collections of objects, which is very useful in OO programming.
Java domain	Classes for Java/COBOL interworking.
Java classes for COBOL support	Java classes that enable you to send messages to COBOL data items.
Net Express GUI classes	Classes for creating and managing graphical applications that exploit the Windows native interface. These include classes for most of the common controls, including dialog boxes, entry fields, pushbuttons, list boxes, and tree views.
Net Express COM	Classes that represent COM documents, servers, clients and controls. It also has classes which wrap common COM interfaces, enabling easy access to frequently used COM functions.
Net Express COM component	Classes for working with COM components.