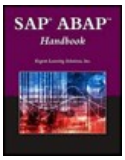

SAP ABAP Handbook

by Kogent Learning Solutions, Inc.

Appendix A: Object Orientation in ABAP

Highlights

Object orientation in ABAP (also known as ABAP Objects) is an extension of the ABAP language that provides the advantages of object-oriented programming (OOP), such as encapsulation, inheritance, and polymorphism. In other words, ABAP Objects, a part of ABAP Workbench, is a programming language used to create and execute program objects in an SAP system. It includes a virtual machine to compile and execute SAP applications. The objects created by using the ABAP Objects language are compatible with the existing programs of the ABAP language. However, the syntax-checking process is stronger in ABAP Objects programs, which ensures that ABAP Objects programs do not use the syntax construction of older programs.

Similar to earlier procedural programming languages, such as COBOL, ABAP was initially developed as a procedural language. However, ABAP has now adapted the principles of object-oriented paradigms with the introduction of ABAP Objects. The object-oriented concepts in ABAP Objects, such as class, object, inheritance, and polymorphism, are essentially the same as those of other modern object-oriented languages, such as C++ or Java.

In this appendix, you learn about the ABAP Objects as an extension of the ABAP language. We begin this appendix by providing an overview of ABAP Objects. Next, you learn the basic concepts of OOP in ABAP, such as classes, objects, encapsulation, and inheritance. You also explore how to declare and implement a class, handle an object, and declare and implement an interface. In addition, you learn how to declare and call methods and constructors in classes and interfaces. The appendix concludes with a discussion on how to define events in a class or interface that triggers the event-handler methods of other classes and interfaces.

Overview of ABAP Objects

ABAP Objects is one of the new-generation languages that have implemented object-oriented features. The concept of the ABAP Objects language has been launched with SAP R/3 Release 4.0. This concept is important from two perspectives. First, it represents the runtime environment of ABAP that indicates that SAP is based on the object-oriented approach. Second, ABAP Objects is an object-oriented extension of the ABAP language, which unites data and functions in objects. ABAP Objects are not only used in existing programs, but it is also used in new ABAP Objects programs.

Prior to SAP R/3 Release 4.0, the nearest equivalent of ABAP Objects were function modules and function groups. The actual task of executing the business logic of a program was performed by function groups. However, the function groups were not used directly; instead they were used through the function modules. For example, you have a function group for processing orders. Now, the function group contains the global data that are actually attributes of the order. In addition, the function modules of the function group represent the actions that have to be performed on the data of the function group. In this way, the entire detail related to the order processing is actually encapsulated in the function group and cannot be addressed directly, but only through the function modules.

Function groups have some limitations, in spite of being object-oriented. For instance, in an ABAP program, you cannot have multiple instances of a single function group, although multiple instances of different function groups can be created. In addition, you need to use the same data structure in all the function modules and function groups. Moreover, changing the internal data structure of a function group affects many users, and it is often difficult to predict the implications.

These limitations can be avoided by using the concept of interfaces and classes in ABAP. The use of an interface or a class ensures that the internal structure of the instance of the interface or class is hidden and can be changed later according to user requirements. The object-oriented paradigm of ABAP Objects allows you to define data and functions in classes instead of function groups. Therefore, by using the ABAP Objects language, you can create an ABAP program that can have various instances or objects of one or more classes. With the introduction of ABAP Objects, an ABAP program does not require loading a function group in the memory of an SAP system to call a function module. Alternatively, the program generates the instances or objects of a class by using the `CREATE OBJECT` statement. Each object has a unique address, which is also called the object reference of that object. You use the object reference of an object to access the object. After giving a brief overview of ABAP Objects, we explain the basic concepts of OOP in ABAP, such as objects, classes, encapsulation, and inheritance.

Explaining the Basic Concepts of OOP in ABAP

OOP is a problem-solving method to find solutions to problems related to programming by using real-world objects. An object in an OOP model is a collection of attributes (data) and methods (functions). Examples of objects in a business

environment include customer, order, and invoice. In SAP Release 3.1 and later, the Business Object Repository (BOR) contains examples of such objects. The object model of ABAP Objects is compatible with the object model of the BOR.

In this section, we describe some fundamental concepts of OOP that also occur in ABAP Objects. These basic terms are:

- Objects
- Classes
- Interfaces
- Encapsulation
- Inheritance
- Polymorphism

Objects

An object is a pattern or instance of a class. For example, Nissan Sentra and Ford Focus are objects, and Car is their class. An object represents a real-world entity, such as a person, place, or a programming entity, such as constants, variables, and memory location. Professors, doctors, and students are examples of real-world entities, whereas hardware and software components of a computer are examples of programming entities. An object has the following three main characteristics:

- Has a state
- May or may not display a behavior
- Has a unique identity

The state of an object can be described as a set of attributes and their values. For example, an account has a set of attributes, such as Account Number, Account Type, Name, and Balance, and the values of each of these attributes. The behavior of an object refers to the changes that occur in its attributes over a period of time. For example, a washing machine is in the static state of behavior when it is switched off. However, it displays a specific behavioral change by washing the clothes when you switch it on.

Each object has a unique identity that distinguishes it from other objects. Two objects may exhibit the same behavior and may or may not have the same state, but they never have the same identity. For example, two persons have the same name, age, and gender but are not identical. The identity of an object never changes throughout its lifetime.

Objects can interact with one another by sending messages. For example, if employee and salary are two objects in a program, the employee object may send a message to the salary object to confirm the tax deduction from the salary. The object that receives the message is called a receiver, and the set of actions taken by a receiver is represented by a method.

Objects contain data and code to manipulate the data. An object can also be used as a user-defined data type with the help of a class. Objects are also called variables of the type class. After defining a class, you can create any number of objects belonging to that class. Each object is associated with the data of the type class with which it has been created.

Classes

Class is a prototype that defines the data and behavior common to all the objects of a specific type. In a class, data is represented by class attributes and the behavior of an instance of the class is provided by class methods. In other words, we can say that classes describe objects.

In the ABAP Objects language, a class is defined by writing the declaration part of the class, and if required, an implementation part of the class. A class can be of two types, global class or local class. A global class can be accessed by all the ABAP programs in an SAP system, while a local class can be accessed within the same program where the class has been defined. In the SAP system, a global class is defined by using the Class Builder tool of ABAP Workbench; however, a local class is defined in an ABAP program.

Interfaces

Interfaces, similar to classes, contain attributes and methods. However, unlike classes, interfaces do not have the implementation part. The methods defined in an interface can be implemented within the implementation of a class. Moreover, interfaces implement the polymorphism feature of OOP in the ABAP Objects language.

Encapsulation

Encapsulation is a feature of OOP that prevents access to non-essential details of a class. For example, when you switch on a television set, it displays the programs being telecast in the form of audio and video. You cannot see the actual complex process of the television, such as how the signals from the cable are being transformed into audio and video. Encapsulation is also referred to as data hiding because it hides important but unnecessary details of an object or a class from the user. Note that the encapsulated data of a class can be accessible outside the class only when the data is used in the functions of the class. However, a limited number of operations can be performed on encapsulated data by executing the functions or methods of the class.

When you use encapsulation, the visibility of the attributes and methods of a class or object are restricted to three basic levels, which are:

- **Public**—Specifies that the members (attributes and methods) of a class are accessible to all users and methods of the class, and any classes inherited from it.
- **Protected**—Specifies that all the members of a class are accessible to all the methods of the class and of classes inherited from it.
- **Private**—Specifies that all the members of a class are accessible only in the methods of the same class.

Note The three levels of visibility of attributes and methods of a class are discussed later in this chapter.

Inheritance

Inheritance is a mechanism that allows a class to inherit the properties of another class. If the *X* class inherits some properties from the *Y* class, we say that *X* has been inherited from *Y*. In this scenario, *Y* is called the superclass or base class of *X*, and *X* is called a subclass or derived class of *Y*, as shown in Figure A.1: The objects of the *X* class have access to attributes and methods of the *Y* class. Objects of subclass *X* can be used where objects of the corresponding superclass *Y* are required. This is because the objects of the subclass *X* share the same behavior as the objects of the superclass *Y*.

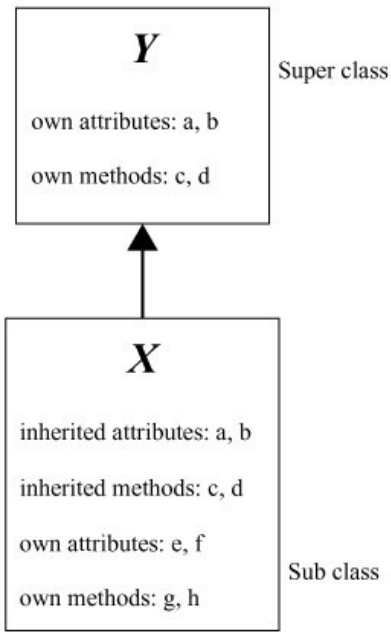

Figure A.1: Showing inheritance between the *X* and *Y* classes

Inheritance enables you to reuse classes in multiple programs or different parts of the same program. This means that we

can add additional features to an existing class without modifying it by deriving a new class from an existing class. In addition, you can define a subclass with some unique features that are different from the features already defined in the corresponding superclass. In addition to its own features, the new class contains the features of the parent class.

In the ABAP Objects language, a subclass contains the attributes and methods of its corresponding superclass, in addition to its own attributes and methods. A subclass derives all the public members (attributes and methods) of its superclass, but not the private members of the superclass. The protected members of a superclass are inherited to a subclass; however, their visibility changes to private.

Polymorphism

Polymorphism is the ability of an object to exist in different forms. The term polymorphism has been derived from the Greek words poly (many) and morphos (forms). In the context of OOP, polymorphism is the ability of a function to behave differently, depending on the context in which it is called.

In ABAP Objects, the names of methods in different classes can be identical, but the methods behave differently in different classes because the implementation of a method (that is, the body of the method) is different in different classes. This is possible in an OOP language either by using the concept of interface or by overriding the methods or by redefining the methods in each class after inheritance. In other words, you can implement polymorphism by using interfaces or overriding methods. Note that an interface enables you to implement methods with the same name in different objects, because the form of a method's name is always the same; however, the implementation of the method is specific to a particular class.

After explaining the basic concepts of OOP in ABAP, we define and implement a class in terms of ABAP Objects.

Defining and Implementing a Class

A class acts as a template for an object. It defines the abstract characteristics of an object, such as attributes, fields, and properties. It also defines the behaviors of objects. For example, the Car class consists of attributes that all cars possess, such as wheels and seats, and the ability in respect of speed mileage (behaviors).

In the ABAP Objects language, global classes are always stored in a class pool in the class library of the SAP repository. All the ABAP programs can access these global classes. A local class, however, is defined in a program, so that it can be accessed in the program itself but not from outside the program. When you use a class in an ABAP program, the SAP system identifies whether or not the class declared in a program is local. If the class is not a local class, the program searches for the required global class in the class pool.

> **Note** A class pool is like a container for a global class and contains exactly one global class. Besides a global class, a class pool can also have global types, local classes, and local interfaces to be used in the global class. In an SAP system, class pools are managed by Class Builder.

The definition of a local class contains ABAP source code between the `CLASS` and `ENDCLASS` statements. Note that a class definition includes the class declaration and, if required, the class implementation. The following syntax shows how to define a local class:

```
CLASS <class_nm> DEFINITION.
...
ENDCLASS.
```

The definition of a local class can contain various components of the class, such as attributes, methods, and events. Moreover, when you declare a method in the class declaration, the method implementation must be included in the class implementation. The following syntax shows how to implement a class:

```
CLASS <class_nm> IMPLEMENTATION.
...
ENDCLASS.
```

Note that the implementation of a class contains the implementation of all its methods.

In ABAP Objects, the structure of a class contains components; that is, attributes, methods, events, types, and constants. Each component of a class is assigned to a visibility section; that is, a public section, protected section, or private section.

Exploring the Components of a Class

The content or definition of a class includes the components of the class. All the components are declared in the

declaration part of the class. These components define the attributes of the objects in a class. All of the components of a class are visible within the class.

All the components of a class can be divided into two categories, instance and static. An instance component exists separately for each object of a class. A static component, on the other hand, is shared by all the objects of a class. This means that only one static component exists for the whole class, regardless of the number of objects in the class.

Attributes

Attributes are data fields of a class that can have any ABAP data type, such as `C`, `I`, `F`, and `N`. They are declared in the class declaration. The data stored in the attributes of a class can determine the state of the objects of the class.

The attributes of a class can be divided into two categories, instance attributes and static attributes. An instance attribute defines the instance- specific state of an object. Instance-specific states are different for different objects. For instance, assume that you are the owner of a company, XYZ. Your company has to interact with other companies to supply goods manufactured in your company. Now, the name and address of all these companies are examples of instance attributes, because the name and address of one company is different from that of the others. An instance attribute is declared by using the `DATA` statement.

Static attributes define a common state of a class that is shared by all the instances of the class. For instance, if we consider the same example of the XYZ company, the terms and conditions to deal with any company regarding the supply of goods are examples of static attributes, because the terms and conditions do not differ for different companies. Note that if you change a static attribute in one object of a class, the change is visible to all the other objects of the class as well. A static attribute is declared by using the `CLASS-DATA` statement.

Methods

A method is a function or procedure of a class that represents the behavior of an object in the class. In a class definition, the methods of the class can access any attribute of the class. The definition of a method can also contain parameters, so that you can supply the values to these parameters when the methods are called, and can also receive the values back from the methods.

The definition of a method is declared in the class declaration and implemented in the implementation part of a class. The `METHOD` and `ENDMETHOD` statements are used to define the implementation part of a method. The following syntax shows how to implement a method:

```
METHOD <meth_nm>.
...
ENDMETHOD.
```

In this syntax, `<meth_nm>` represents the name of a method. Note that you can call a method by using the `CALL METHOD` statement.

Similar to other components of a class, the methods of a class can also be divided into two basic categories, instance methods and static methods. A method that can access all the attributes and trigger all the events of a class is called an instance method. An instance method is declared by using the `METHODS` statement.

Static methods of a class, on the other hand, can access only static attributes and trigger static events. They are declared by using the `CLASS-METHODS` statement.

In addition to instance and static methods, constructors are special methods that are called automatically, either while creating an object or accessing the components of a class. Constructors are again of two types, instance and static. An instance constructor is declared by using the `CONSTRUCTOR` statement. It can access all the attributes and can trigger all the events of a class. A static constructor, on the other hand, is declared by using the `CLASS_CONSTRUCTOR` statement. It can access the static attributes and static events of a class.

Events

An event is a set of outcomes that are defined in a class to trigger the event handlers in other classes. In addition, when an event is triggered, any number of event handler methods can be called. The link between a trigger and its handler method is decided dynamically at runtime.

In a normal method call, a calling program determines which method of an object or a class needs to be called, and when. However, in the case of event handling, the handler method determines the event that needs to be triggered. This is

because a fixed handler method is not registered for every event.

An event of a class can trigger an event handler method of the same class by using the RAISE EVENT statement. In addition, for an event, the event handler method can be defined in the same or different class by using the FOR EVENT clause, as shown in the following syntax:

```
FOR EVENT <evt_nm> OF <class_nm>.
```

Similar to the methods of a class, an event can also have parameter interface but has only output parameters. The output parameters are passed to the event handler method by the RAISE EVENT statement, which receives them as input parameters. An event is linked to its handler method dynamically in a program by using the SET HANDLER statement.

Events are also of two types, instance events and static events. An instance event can be triggered only from an instance method. It is declared by using the EVENTS statement. A static event however, can be triggered only from a static method. It is declared by using the CLASS-EVENTS statement. Note that an event and its handler can be an object (in the case of instance events) or a class (in the case of static events). In addition, when an event is triggered, the corresponding event handler methods are executed in all the registered handling classes.

Types

In a class, you can create your own ABAP data types by using the TYPES statement. Types are not instance-specific and exist only once for all of the objects in a class.

Constants

Constants are static values for the attributes of a class. These values are specified at the time of declaring the attributes in a class. Constants are declared by using the CONSTANTS statement. Moreover, constants are not instance-specific but exist only once for all of the objects in a class.

Note All the components of a class can also be declared inside the declaration of an interface.

Next, we explain the visibility sections for the components of a class.

Visibility Sections in a Class

In a class definition, each component of the class is assigned to one of the three visibility sections, PUBLIC SECTION, PROTECTED SECTION, or PRIVATE SECTION, to define the external access of the component outside the class. The following syntax shows the three visibility sections in a class:

```
CLASS <class_nm> DEFINITION.
PUBLIC SECTION.
...
PROTECTED SECTION.
...
PRIVATE SECTION.
...
ENDCLASS.
```

In this syntax, <class_nm> represents the name of a class and the CLASS and ENDCLASS statements represent the start and end of the class definition. The three visibility sections in a class are represented by the PUBLIC SECTION, PROTECTED SECTION, and PRIVATE SECTION keywords.

Note that the public components of a global class may not be changed. Moreover, after defining the visibility of an attribute, you can protect it from any external change by using the READ-ONLY clause.

Listing A.1 shows an example of a class along with its components and visibility sections:

Listing A.1: Showing the components and visibility sections in a class

```
CLASS NUM_COUNT DEFINITION.
  PUBLIC SECTION.
    METHODS: SET_NUM IMPORTING VALUE (SET_VALUE)
             TYPE I,
             INCREMENT_NUM,
             GET _ NUM EXPORTING VALUE (GET _ VALUE)
             TYPE I.
```

```
  PRIVATE SECTION.
    DATA NUM TYPE I.
ENDCLASS.
CLASS NUM_COUNT IMPLEMENTATION.
  METHOD SET_NUM.
    NUM = SET_VALUE.
  ENDMETHOD.
  METHOD INCREMENT_NUM.
    ADD 1 TO NUM.
  ENDMETHOD.
  METHOD GET_NUM.
    GET_VALUE = NUM.
  ENDMETHOD.
ENDCLASS.
```

In this example, NUM_COUNT is a class containing three public methods: SET_NUM, INCREMENT_NUM, and GET_NUM. Each of these methods accesses a private field, Num, of the I type. The SET_NUM method has an import parameter, SET_VALUE, while the GET_NUM method has an export parameter GET_NUM. Note that the methods declared in the definition of the NUM_COUNT class are used in the implementation of the class, but the NUM field is not. This is because the NUM field is defined in the private section of the NUM_COUNT class.

Handling the Objects

In an ABAP program, an object is accessed by using the object references, which are pointers to objects. In the ABAP Objects language, object references are stored in reference variables. In ABAP, reference variables are treated as the other elementary data objects, which means that a reference variable can be a component of a structure or an internal table, or it can refer to itself.

In ABAP Objects, the object reference can be of the type class reference or interface reference. A class reference is defined by using the TYPES or DATA statement along with the following syntax:

```
... TYPE REF TO <class_nm>
```

A class reference allows you to create an instance or object of the corresponding class and to access a component of the class, as shown in the following syntax:

```
cref->comp
```

In this syntax, cref represents class reference and comp represents a component of the class.

To create an object for a class, you need to declare a reference variable of the class. Use the following syntax to create an object of a class by using the CREATE OBJECT statement:

```
CREATE OBJECT <cref>.
```

In this syntax, <cref> represents a reference variable of a class. The CREATE OBJECT statement creates an object of a class with the <cref> class reference.

An object exists in a program until at least one reference points to it, or at least one method of the object is registered as an event handler. When no reference of the object is used in a program and none of the methods of the object is registered as event handler, the object is deleted by the automatic memory management (garbage collection) process.

Declaring and Implementing Interfaces

Similar to classes in ABAP Objects, interfaces act as data types for objects. The components of interfaces are same as the components of classes; that is, attributes, methods, events, types, and constants. However, the declaration of an interface does not include the visibility sections, unlike the declaration of classes. This is because the components defined in the declaration of an interface are always integrated in the public visibility section of the classes.

Interfaces are used when two similar classes have a method with the same name, but the functionalities of these methods are different from each other. Interfaces might appear similar to classes; however, unlike classes, the functions defined in an interface are implemented in a class to extend the scope of that class. Moreover, interfaces, along with the inheritance feature, provide a base for polymorphism, because a method defined in an interface can behave differently in different classes.

Similar to global and local classes, interfaces can be classified as global or local. A global interface can be used in any program, while a local interface can be used only in the same program in which it is declared. Similar to a class, you can create a global interface by using Class Builder. Use the PUBLIC clause to identify an interface as a global interface. A global interface is stored in an interface pool, which cannot contain any local type declaration. Note that the interface pool is used to either implement an interface defined in it or to create reference variables of the interface type defined in it.

You create a local interface in an ABAP program by using the following syntax:

```
INTERFACE <intf_nm>.
  DATA ...
  CLASS-DATA ...
  METHODS ...
  CLASS-METHODS ...
  ...
ENDINTERFACE.
```

In this syntax, <intf_nm> represents the name of an interface. The DATA and CLASS-DATA statements can be used to define the instance and static attributes of the interface, respectively. In addition, the METHODS and CLASS-METHODS statements can be used to define the instance and static methods of the interface, respectively.

Unlike classes, the definition of an interface does not include the implementation class. Therefore, it is not necessary to add the DEFINITION clause in the declaration of an interface. Note that all the methods of an interface are abstract. They are fully declared, including their parameter interface, but not implemented in the interface. All the classes that want to use an interface must implement all the methods of the interface; otherwise, the class becomes an abstract class.

To use an interface in a class, use the following syntax in the implementation part of the class:

```
INTERFACES <intf_nm>.
```

In this syntax, <intf_nm> represents the name of an interface. Note that this syntax must be used in the public section of the class that wants to use the <intf_nm> interface.

The following syntax is used to implement the methods of an interface inside the implementation of a class:

```
METHOD <intf_nm_meth>.
...
ENDMETHOD.
```

In this syntax, <intf_nm_imeth> represents the fully declared name of a method of the <intf_nm> interface.

Listing A.2 shows an example to declare and implement an interface:

Listing A.2: Declaring and implementing an interface

```
INTERFACE my_interface.
  METHODS message.
ENDINTERFACE.

CLASS num_counter DEFINITION.
  PUBLIC SECTION.
    INTERFACES my_interface.
    METHODS add_num.
  PRIVATE SECTION.
    DATA num TYPE I.
ENDCLASS.

CLASS num_counter IMPLEMENTATION.
  METHOD my_interface~message.
    WRITE: / 'The number is', num.
  ENDMETHOD.
  METHOD add_num.
    ADD 1 TO num.
  ENDMETHOD.
ENDCLASS.

CLASS car DEFINITION.
```

```
  PUBLIC SECTION.
    INTERFACES my_interface.
    METHODS speed.
  PRIVATE SECTION.
    DATA wheel TYPE I.
ENDCLASS.
CLASS car IMPLEMENTATION.
  METHOD my_interface~message.
    WRITE: / 'The number of wheels in the car is', wheel.
  ENDMETHOD.
  METHOD speed.
    ADD 10 TO wheel.
  ENDMETHOD.
ENDCLASS.
```

In Listing A.2, `my_interface` is the name of an interface that contains the message method. Next, two classes, `num_counter` and `car`, are defined and implemented. Both these classes implement the message method, in addition to the specific methods that define the behavior of their respective instances, such as the `add_num` and `speed` methods. Note that the `add_num` and `speed` methods are specific to the respective classes and are not related to the `my_interface` interface.

Now, let's explore how to declare, implement, and call the methods in classes and objects.

Declaring and Calling Methods

As stated earlier, methods are internal procedures in a class, which define the behavior of an object or instance of the class. In the ABAP Objects language, instance methods are declared by using the `METHODS` statement. The following syntax is used to declare an instance method:

```
METHODS method_nm IMPORTING [VALUE(]i1 i2 ... [)] TYPE type
[OPTIONAL]...
EXPORTING [VALUE(]e1 e2 ... [)] TYPE type ...
CHANGING [VALUE(]c1 c2 ... [)] TYPE type [OPTIONAL]...
RETURNING VALUE(r)
EXCEPTIONS exc1 exc2 ... .
```

Static methods are declared by using the `CLASS-METHODS` statement. The following syntax is used to declare static class methods:

```
CLASS- METHODS method_nm IMPORTING [VALUE(]i1 i2 ... [)]
T  YPE type [OPTIONAL]...
EXPORTING [VALUE(]e1 e2 ... [)] TYPE type ...
CHANGING [VALUE(]c1 c2 ... [)] TYPE type [OPTIONAL]...
RETURNING VALUE(r)
EXCEPTIONS exc1 exc2 ... .
```

In this syntax, when a method (instance method or static method) is declared, its parameters are declared by using the `IMPORTING` (input parameter), `EXPORTING` (output parameter), `CHANGING` (input/output parameter), and `RETURNING` (return code) clauses. You can also define whether a parameter needs to be passed by reference or value (`VALUE`), the type of parameter (`TYPE`), and whether the parameter is optional or default, by using the `OPTIONAL` or `DEFAULT` clause, respectively. Note that, unlike function modules, the default way of passing a parameter in a method is by reference. The `VALUE` clause is used to pass a parameter by value. The `RETURNING` clause is used to return a value, which must always be passed explicitly as a value. This is suitable for methods that return a single output value. Note that you cannot use the `EXPORTING` or `CHANGING` clauses with the `RETURNING` clause. Moreover, the `EXCEPTIONS` clause is used to define exception parameters, similar to function modules, which allow users to handle errors when the method is executed.

Next, use the following syntax to implement a method in the implementation part of a class:

```
METHOD method_nm.
...
ENDMETHOD.
```

In this syntax, the `method_nm` expression represents the name of a method. The implementation code of a method must be written between the `METHOD` and `ENDMETHOD` statements. You should note that the implementation part of a method

does not include the parameters of the method, as the parameters are defined only in the method's declaration part. The parameters declared in a method act as local variables in the implementation of the method. However, you can define additional local variables in the implementation of a method by using the DATA statement. Moreover, you can use the RAISE and MESSAGE RAISING statements in the implementation of a method to handle errors.

The following syntax is used to call a method:

```
CALL METHOD method_nm | ref->method_nm | class_nm => method_
nm EXPORTING i1 = f1 i2 =f2 ...
    IMPORTING e1 = g1 e2 =g2 ...
    CHANGING c1 = f1 c2 =f2 ...
    RECEIVING r = h
    EXCEPTIONS e1 = rc1 e2 =rc2 ...
```

In this syntax, the CALL METHOD statement and its parameters are used to call a method, depending on the method declaration. The following syntax is used to call a method of the same class directly by using its name, method_nm:

```
CALL METHOD method_nm ...
```

You can call a method from outside a class by using the following syntax:

```
CALL METHOD ref->method_nm ...
```

In this syntax, ref is a reference variable whose value points to an instance of a class. Outside the class, a method can be called based on the visibility of the method. The visibility of a method, however, is decided by you, when you declare a method.

Visible instance methods can also be called from outside a class by using the following syntax:

```
CALL METHOD class_nm => method_nm ...
```

In this syntax, class_nm is the name of the relevant class.

When a method is called, all non-optional input parameters are passed by using the EXPORTING or CHANGING clause in the CALL METHOD statement. You can import the output parameters by using the IMPORTING or RECEIVING clause and can handle exceptions by using the EXCEPTIONS clause.

The methods defined in an interface can also have parameters. Consider the following expression:

```
... Formal parameter = Actual value or parameter
```

In this expression, the interface parameters, also called formal parameters, are specified at the left side of the equals (=) sign. However, the actual values or parameter for the interface parameters are specified at the right side of the equals sign.

Note that if the call to a method contains only a single import parameter, you can use the following shortened syntax form of the method call:

```
CALL METHOD method (act_par).
```

In this syntax, act_par represents the actual parameter, which is passed to the input parameters of the method being called.

Similarly, if the call to a method contains only import parameters, you can use the following shortened syntax form of the method call:

```
CALL METHOD method (frm_par1 = act_par 1 frm_par2 = act_
par 2 ...).
```

In this syntax, act_par 1, act_par 2,... act_par n represent the actual parameters and frm_par 1, frm_par 2,... frm_par n represent formal parameters. Each actual parameter is assigned to the corresponding formal parameter.

Next, we explore how to declare and call a constructor in ABAP Objects.

Declaring and Calling Constructors

Constructors are called automatically when you create an object or access the components of a class for the first time. However, constructors are not called by using the CALL METHOD statement, as in the case of a normal method call. The

instance constructor of a class is the predefined instance method, called CONSTRUCTOR. An instance constructor is always declared in the public section of a class. Use the following syntax to declare an instance constructor:

```
METHODS CONSTRUCTOR
    IMPORTING.. [VALUE(]<par 1> <par 2>...<par n>[)] TYPE type
    [OPTIONAL]..
    EXCEPTIONS.. <e 1> <e 2>...<e n>.
```

The static constructor of a class is the predefined static method, called CLASS_CONSTRUCTOR. Similar to an instance constructor, a static constructor is also declared in the public section of a class. Use the following syntax to declare a static constructor:

```
CLASS-METHODS CLASS_CONSTRUCTOR.
```

A constructor is implemented in the implementation section of a class, just like any other method. An instance constructor is called once for each instance of the class, after the object has been created in the CREATE OBJECT statement. You can pass the values to the input parameters of an instance constructor and can handle its exceptions by using the EXPORTING and EXCEPTIONS clauses.

A static constructor, however, has no parameters. A static constructor is called once for each class, before the class is accessed for the first time. Therefore, a static constructor cannot access the components of its class.

Now, let's explore another important component of a class or an interface; that is, events.

Working with Events in ABAP Objects

In the ABAP Objects language, certain methods act as triggers or events that are handled or reacted by certain other methods, called handlers or handler methods. This means that handler methods are executed when associated events occur. The link between a trigger and its handler method is decided dynamically at runtime.

Triggering an Event

An event is triggered by declaring an event in the declaration part of a class or interface. The event is triggered in one of the methods of the class associated with the event. The EVENTS statement is used to declare an instance event, and the CLASS-EVENTS statement is used to declare a static method. Use the following statement to declare an instance event:

```
EVENTS <evt_nm> EXPORTING... VALUE(<e 1> <e 2>…<e n>) TYPE
type [OPTIONAL]..
```

Use the following statement to declare a static event:

```
CLASS-EVENTS <evt_nm> EXPORTING... VALUE(<e 1> <e 2>…<e n>)
TYPE type [OPTIONAL]..
```

When an event is declared, you can use the EXPORTING clause to specify the parameters that are passed to the event handler. The parameters are always passed by value. Moreover, instance events always contain the SENDER implicit parameter of the type of a reference to the class or the interface in which the event is declared.

After declaring an event in the declaration part of a class or interface, the event is executed or triggered as a method of a class by using the RAISE EVENT statement. In case of an instance event, the event can be triggered by any method in the class; however, in case of a static method, the event can be triggered by any static method. Use the following statement to trigger an event in a method:

```
RAISE EVENT <evt_nm> EXPORTING... <e 1> = <f 1> <e 2> =
<f 2>...<e n> = <f n>
```

In this syntax, <e 1>, <e 2>...<e n > represent formal parameters, which are non-optional, and <f 1>, <f 2>...<f n > represent actual parameters, corresponding to each formal parameter. Note that the ME self-reference is automatically passed to the SENDER implicit parameter.

Handling an Event

When an event is triggered, it calls its event handler. For this, you must define an event-handler method and register it for the event at runtime. An event handler method can be defined either in the same class of the event or in a different class. Use the following syntax to declare an event-handler method corresponding to an instance event of a class or interface:

```
METHODS <method_nm> FOR EVENT <evt_nm> OF <cif> IMPORTING..
```

```
<e 1> <e 2>…<e n>
```

Use the following syntax to declare an event-handler method corresponding to a static event:

```
CLASS-METHODS <method_nm> FOR EVENT <evt_nm> OF <cif>
IMPORTING.. <e 1> <e 2>…<e n>
```

In both syntaxes, `<method-nm>` represents a method and `<evt_nm>` represents an event declared in the `<cif>` class or interface. The handler method contains the formal parameters defined in the declaration of the `<evt_nm>` event. It is not necessary for an event-handler method to use all the parameters, which are passed in the `RAISE EVENT` statement. Note that when an event-handler method is declared in the same class of the event, the instances of such class or the class itself handle an event, `<evt_nm>`, which is triggered in a method.

As stated earlier, a handler method can react to an event after registering it for the event at runtime. The following syntax is used to register an event handler method:

```
SET HANDLER... <h 1> <h 2>...<h n>... [FOR]...
```

In this syntax, the `SET HANDLER` statement is used to link a list of handler methods with the corresponding triggers or events. As we know, an event can be either instance or static. The `FOR` clause is used in the `SET HANDLER` statement to register an event-handler method corresponding to an instance event. Use the following syntax to handle an instance event by using a reference variable, `<ref>`:

```
SET HANDLER... <h 1> <h 2>…<h n>...FOR <ref>.
```

Use the following syntax to register the handler for all instances that can trigger the event:

```
SET HANDLER... <h 1> <h 2>…<h n>...FOR ALL INSTANCES.
```

In this syntax, the registration of handler methods applies even to triggering instances that have not yet been created when you register the handler.

The following syntax is used to handle a static event without using the FOR clause:

```
SET HANDLER... <h 1> <h 2>…<h n>...
```

In this syntax, the registration of handler methods applies to the entire class as well as all those classes that implement the interface that contains the static event. When a static event of an interface has to be handled by a handler method, the registration also applies to classes that are not loaded until the handler has been registered.

Handler methods are executed in the order in which they were registered. Because event handlers are registered dynamically, the order in which they would be processed is not known. In fact, all event handlers can even be executed simultaneously.

Furthermore, after the execution of the `RAISE EVENT` statement, all registered handler methods are executed before the next statement is processed (synchronous event handling). If an event-handler method itself triggers events, its handler methods are executed before the original handler method executes. An event handler can be nested to a maximum of 64 levels to avoid the possibility of falling into an infinite loop.