

R/3[®] System

Object-Oriented Concepts of ABAP[®]

©Copyright 1997 SAP AG. All rights reserved.

No part of this brochure may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft®, WINDOWS®, NT®, EXCEL® and SQL-Server® are registered trademarks of Microsoft Corporation.

IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, und OS/400® are registered trademarks of IBM Corporation.

OSF/Motif® is a registered trademark of Open Software Foundation.

ORACLE® is a registered trademark of ORACLE Corporation, California, USA.

INFORMIX®-OnLine *for SAP* is a registered trademark of Informix Software Incorporated.

UNIX® and X/Open® are registered trademarks of SCO Santa Cruz Operation.

ADABAS® is a registered trademark of Software AG

SAP®, R/2®, R/3®, RIVA®, ABAP®, SAPoffice®, SAPmail®, SAPaccess®, SAP-EDI®, SAP ArchiveLink®, SAP EarlyWatch®, SAP Business Workflow®, SAP Retail®, ALE/WEB®, SAPTRONIC® are registered trademarks of SAP AG.

Contents

Object-Oriented Concepts of ABAP	3
Goals and Motivation	3
An Overview of Object-Oriented Concepts	4
Attributes in Detail	6
Methods in Detail	6
Events in Detail	7
Classes in Detail	8
Interfaces in Detail	10
References to Objects and Interfaces	12
The Object Life Cycle	13
Local and Global Classes and Interfaces	14
Summary	14

Object-Oriented Concepts of ABAP

As part of SAP's long-standing commitment to object technology, Release 4.0 of R/3 will contain object-oriented enhancements to the ABAP programming language. SAP's object strategy is based on *SAP Business Objects* and now covers *modeling, programming, interfacing, and workflow*. By using principles like *encapsulation, inheritance, and polymorphism*, the object-oriented extensions of ABAP will support real object-oriented development. This will result in improvements in the areas of *reusability, maintenance, and quality of code*. SAP offers an evolutionary approach toward objects which leverages SAP's own and its customers' investments in existing business processes, functionality and data.

Abstract

attribute, class, COM/DCOM, CORBA, encapsulation, event, garbage collection, inheritance, interface, method, object, polymorphism, SAP Business Object

Key words

Goals and Motivation

The implementation of software systems that meet rapidly evolving business requirements is eased by business objects that mirror real business entities and, at the same time, are represented by true software objects. Instead of a purely technical view, SAP employs object technology from a *business point of view*, based on more than 20 years of experience in developing enterprise business systems. The main emphasis is on the design and implementation of *business processes based on business objects*. To support this strategy, SAP offers a comprehensive and - even more important - *business driven* approach with the Business Framework. After developing object-oriented solutions for modeling, interfacing, and workflow in earlier R/3 releases, SAP's enhancements to ABAP in Release 4.0 open the way for object-oriented programming.

The Objective

Programmers will benefit from the enhancements to ABAP through *increased productivity* as well as through *higher quality* and *easier maintenance* of code. The *increased reusability* of program components supported by business-oriented modeling tools means that R/3 is able to make the quality and productivity of application development even better.

Applications developers are at their most productive when they are able to concentrate their efforts on solving the application problem itself rather than on other aspects of program development. The incorporation of object-oriented concepts into ABAP moves the language a step closer to this goal. SAP's main priorities here are to exploit the following advantages offered by the object-oriented development:



- ❑ **Clear and understandable systems through structuring:** Structuring at all levels results in clarity throughout the whole system, both within individual components and between different components.
- ❑ **Maintainability through encapsulation and decoupling:** The encapsulation of data and functionality within objects means that it is much easier to make the necessary changes to program components. Each object has a well-defined interface, and can only be accessed through this interface. This conceals the internal structure and allows changes to be made without affecting other program components.
- ❑ **Faster and more standardized implementation through reuse:** The reuse of designs and program components becomes simpler. The object-oriented enhancements to ABAP allow, for example, the development of object-oriented frameworks.
- ❑ **Interfaces to standards like OLE/DCOM and CORBA** become easier to use and to develop. It will now become even easier to develop **SAP Business Objects** and to support **SAP Business Workflow** and future **GUI developments**.

By leveraging these advantages, the object-oriented enhancements to ABAP contribute significantly to the realization of SAP's object strategy described above. This paper describes the programming model of 'ABAP with objects' without referring to the syntax. It provides the basic information required for developing object-oriented applications with ABAP.

An Overview of Object-Oriented Concepts

Objects

The concept of an object is central to the object-oriented development. The idea of objects is that the entities involved in a problem should be represented as far as possible one-to-one by objects in the proposed solution. An **object** is a self-contained entity having a **state**, a **behavior** and an **identity**.

Public, protected, and private components

Since objects are self-contained entities, many of their aspects are not visible from the outside. For this reason, it is important to distinguish between the outer view and the inner view of an object. Depending on its visibility, an object component is designated as *public*, *protected* or *private*.

- Attributes** ❑ Objects have **attributes**. These describe the current state of the object.
- Methods** ❑ The behavior of an object is described by its **methods**.
- Identity** ❑ Each object has a unique **identity** which never changes. The identity of an object allows it to be distinguished from other objects which have the same set of attributes and the same attribute values.
- Events** ❑ A further feature of ABAP objects is that they can both raise and handle **events**. When one object raises an event, one or more other objects can be informed that this event has occurred, and can then react as

appropriate. An event can also occur without other objects being informed of this.

There are three different ways in which objects can interact with each other.

- ❑ **Direct data access:** An object has direct read and write access to the visible attributes of other objects.
- ❑ **Calling a method directly:** When a method is called directly, a unique object is called (known as the *server* or *supplier*) and this object is known to the calling object (known as the *client*).
- ❑ **The publish-subscribe mechanism:** If an event is raised while a method is being executed, all objects currently registered as being interested in this event (the “subscribers”) will be informed of it. It is therefore possible for an event to occur and for no object to handle it, simply because no object has subscribed to the event. The object that raised the event does not know which objects have subscribed to this event.

Accessing visible attributes

Calling server methods directly from the client

Publishing an event

Classes

The definition of a class contains the specifications of the interfaces and the structure of the objects in that class. The objects belonging to a class are also referred to as the **instances** of the class.

Instances

A new class can be defined as a specialization of an existing class. The more specialized class is referred to as a **subclass**, the more generalized class as the **superclass**. Any statements that are true for the outward behavior of the superclass are also true for the subclass. In particular, all of the public attributes and methods of the superclass will also be found in the subclass and will have the same semantics as defined for those in the superclass. Subclasses also “inherit” the implementations of all the superclass methods. These methods may, however, be overridden within the subclass.

Generalization and specialization

Interfaces

Two objects that have the same interface may be treated identically by clients even though each may have been implemented quite differently from the other. ABAP provides the **interface** construct for this purpose. The use of this construct means that two objects may offer a common interface even though they do not have a common superclass. A client can therefore treat two objects that have the same interface in the same way, without insisting that both objects be instances of the same superclass.

Interfaces are independent of the implementation

The definition of a class can specify that the class implements one or more interfaces. Each instance of this class then possesses all the attributes of these interfaces, and can be called using the associated methods and can raise the events defined in the interfaces. A method body (the implementation of the method) has to be specified for each method contained in the interface in the implementation of the class.

Classes implement interfaces

Interfaces can **contain** other interfaces. Classes that implement a composite interface have to provide implementations for all the contained interfaces.

Interfaces that contain other interfaces



Attributes in Detail

- Attributes store the object state** Objects can possess **attributes** that contain information about their internal state. Attributes can have one of the data types available in the existing ABAP language (for example, integer, character field, structure or internal table), and also the new object or interface reference type.
- Public, protected and private attributes** Attributes are variables that are local to the object and are therefore not normally visible from the outside. In certain cases, however, it can be useful to make certain attributes visible, so that they can also be used as variables from the outside.
- The components of an object that are only visible within a class definition are called *private*. Components of objects that are in addition only visible in the definitions of the direct and indirect sub-classes are called *protected*. All other components are *public*. These are visible to all 'clients' of the object, i.e. other program components and other class definitions.
- READ-ONLY attributes** Public attributes can be flagged as not modifiable (READ-ONLY) from outside the object.
- Virtual attributes** Attributes can also be flagged as **virtual**. Virtual attributes can be accessed in the same way as normal attributes, but the implementing class then has the freedom to use an access method instead of making a direct attribute access. For this reason, virtual attributes cannot be referenced with field symbols or passed as reference parameters to a routine (METHOD, FUNCTION or FORM). Using virtual attributes allows you, for example, to postpone the decision as to whether attribute accesses should be replaced by access routines until later, without having to make changes to all the program components that access the attributes concerned.
- Moreover, virtual attributes in ABAP enable you to determine at runtime whether the current value of an attribute is valid. If the value is valid, it can be accessed directly, if not, it must be recalculated by a method call. With virtual attributes, a class can defer recalculation of a value until it is really needed ("*lazy evaluation*").
- Class attributes** **Class attributes** can be defined for a class. These are attributes that are common to all the instances of both the class and all its subclasses. Class attributes can be specified as public, protected or private, READ-ONLY and/or virtual, just as for the instance attributes.
- Constants** In addition to variable class attributes, ABAP also allows **constants** to be defined that are common to all the instances of a class. These can be regarded as class constants.

Methods in Detail

- Accessing the local state of objects and performing inter-object communication** Objects are able to perform operations. The **methods** of an object are the means for doing this. A method always operates on a particular instance of a class. It can read and change the state, i.e. all attributes (public, protected



and private) of the instance, and can interact with other objects. Similarly to function modules, methods can have parameters and can pass back a return value.

Methods are also used to specify the way in which objects handle events.

Each method in a class must be uniquely identifiable using its name. Method names may not be “overloaded”.

Class methods are methods that can access class attributes only. Since these methods do not access the instance-specific attributes, they can be used even if no instance of this class has been created yet or if no instance is known. As with instance-specific methods, class methods can be declared as either public, protected, or private. However, class methods cannot raise events.

Events in Detail

Events are characterized by the fact that they occur at a particular point in time. When an event is raised, other things can occur as a consequence of the event. For example, one consequence of an event could be that a process¹ is started or that other events are raised.

Examples of events are changes in the state of an object, such as “posting canceled” or the creation of a new object such as “new account created”, that have to be made known to all other interested objects.

Objects in ABAP can raise events. Other objects can react to these events in an appropriate way. An object that raises an event makes no assumptions about the number of objects that will handle the event, when they will do so or how. The raising object therefore does not generally know which objects will handle the event either. This means that the event concept of ABAP must be distinguished from a callback concept where exactly one object is expected to react in a specific way.

Before an object is in a position to react to the events raised by another object, at least one of its methods must be registered as an **event handling method** for that object. An event handling method may be registered for the events of a single object, multiple individual objects, all instances of a class, or all instances that implement a certain interface.

If an object wishes to cease handling the events raised by another object at any time, it can cancel the registrations of its methods.

If an object raises an event, every event handling method currently registered for this kind of event is informed of it. The object that raised the

Event handling

Method names cannot be overloaded

Class methods

Events

Example

Undefined event handling

Registering event handling methods

De-registering

Raising events

¹ Processes can be characterized as existing over a particular period of time.



event can pass parameters to the event. If there are several handling methods registered for the event, the order in which they are processed will be determined by the system.

Calling event handling methods directly

The event handling methods can also be called directly. There are two possible reasons for doing this. One is to inform a non-registered object that an event has been raised so that the object can handle the event just as if it had received the event directly. The other reason is to allow the raising of events to be reproduced.

Event parameters

Events can have parameters just like methods. However, from the point of view of the event handling method, all event parameters are input parameters. No provision is made either for reference parameters or for a return value.² The event handling method can find out the object that raised the event.

Classes in Detail

A system will often require several objects that possess the same attributes and methods and can raise the same events. These objects will often differ only in their current state. For this reason objects are defined in classes rather than in a one-off fashion. The description of a class contains the declarations of both the public and the private components of the objects and specifies their implementation. The definition of a class contains specifications of the attributes and methods of the individual instances and also of the class attributes and class methods. It describes the events that can be raised by instances of the class and the interfaces that can be used to provide access to the instances. It also specifies the events which can be handled by instances of the class. A class can also be declared as a subclass of another class.

Classes not instances of meta-classes

Although it is possible to define class attributes and class methods, classes cannot be seen as objects with an existence of their own nor can they be referenced as objects. In other words, classes are not instances of meta-classes.

Classes the smallest unit of encapsulation

The smallest unit of encapsulation is the class and not the object. Therefore all methods defined in a class can use the private attributes and methods of all the instances of the class – and not just those of the current instance.³ There is, however, one restriction to this: The methods of a subclass do not have access to the private methods and attributes that are defined in a superclass – unless the superclass has explicitly allowed this form of access (see the friend concept).

² This leaves open the options of informing more than one event handling method that an event has been raised and of letting events be processed asynchronously. These parameter restrictions could perhaps be relaxed in the future for events for which there is always just one event handling method, or when synchronous handling is desired.

³ This is in contrast to the situation in, for example, Smalltalk 80, in which a method has access to the private attributes of the current server only and to the public attributes of all other objects.

Generalization and Specialization

ABAP allows classes to be declared as **direct subclasses** of a class that is already defined - their **direct superclass** (simple inheritance). Considering this 'direct' superclass-subclass relationship in a transitive fashion leads to the general superclass-subclass relationship: The **superclasses** of a class include not just the direct superclass but also all the indirect superclasses, in other words, the superclasses of the direct superclass. Similarly, the **subclasses** of a class include all the direct subclasses and all of their subclasses.

Simple inheritance

An object that is an instance of a subclass is simultaneously a (specialized) instance of each of its superclasses (→ **specialization**). Therefore clients can use these objects just as they would use objects that are instances of any of their superclasses⁴.

Identical behavior

The attributes, methods, events and supported interfaces are also implicit attributes, methods, events and supported interfaces of each of their subclasses. All the relevant properties, such as the meaning and types of the attributes and the meaning and parameters of methods and events are defined by the superclass. Only the implementation of methods can be overridden in sub-classes. A so-called scope resolution operator ensures that access is possible even to methods that have been redefined.

Overriding methods

Subclasses are also able to define additional attributes, methods and events and to support additional interfaces.

Additional components

The components of subclasses live in the same name space as the components of their superclasses. This means that once a name has been used for a public (or protected) component in a superclass, it cannot be used again in a subclass.

Same name space for components

Classes Implement Interfaces

Classes can implement one or more interfaces (see Interfaces below). All objects that implement the same interfaces can be treated identically by clients, regardless of the class to which they belong, i.e. a client can access all these different classes by the same type of interface reference. An interface reference only exposes the components defined directly in that interface.

Implementing an interface just involves defining a method body for each method contained in the interface. If a class implements composite interfaces, then it must also implement the interfaces that are contained in the composite interfaces. The attributes defined in the interface are automatically added to the instances of the class that implements these interfaces.

⁴ The language guarantees that subclasses can syntactically be used like any of their superclasses. The implementation of the subclass must of course guarantee that the subclass also *behaves* like a special case of the superclass (semantic specialization).



Embedding interface components in the name space of the class

Avoiding Naming Conflicts and Aliasing

Each interface and each class definition has a separate name space for their components (attribute, method and event definitions). If classes implement interfaces, the name spaces of the implemented interfaces are not incorporated in the name space of the class, i.e. the components of the implemented interfaces don't automatically appear on the 'top level', i.e. as components of the class. The same rule applies when interfaces contain other interfaces (see below). This eliminates naming conflicts that could arise when classes implement interfaces and for composite interfaces. For example, no naming conflict can occur between identically named attributes and methods in different interfaces - even if a class implements several of these interfaces. As a result, each class can support any number of interfaces at any time, without the need ever to change the clients of this class to resolve name conflicts. The further development of new and existing classes, and interfaces thus remains unlimited and guarantees the independent further development of application components.

Through the explicit definition of *aliases* for components of supported interfaces, a class can declare specific interface components in its own name space.

The Friend Concept

Friend: A directed relationship

Sometimes two classes need to cooperate so closely, that it is convenient to allow one class to access another class's private components. This special relationship is expressed by the 'Friend' concept and it has to be agreed upon by both classes.

The friendship relationship is a directed relationship. This means that if class A offers friendship to class B, i.e. allows B access to its private components, it doesn't follow that class B has to offer friendship to class A.

Interfaces in Detail

Classification and use independent of implementation

The definition of **interfaces** allows objects to be used quite independently of their implementation. Common aspects of the interfaces of two objects can be expressed, even though the objects do not have a common superclass.

Example

The `IPSPrintableObject` interface could, for example, specify that all classes that implement this interface must define the `generatePostscript()` method. Then all instances for which the `IPSPrintableObject` interface is available, whether posting documents, sales orders, ABAP program text or graphic modeling documents, can be addressed as `IPSPrintableObjects` and instructed to generate a Postscript document that can be output on any Postscript printer.

A description of possible interaction

Unlike class definitions, interface definitions do not contain an implementation section. An interface definition only describes the interaction that is possible with an object of any class that supports the interface. Clearly only the public components of objects can be specified in interface definitions, namely their public attributes, their public methods and the events that the object can raise.

Combining Interfaces

Interfaces are usually developed in a bottom-up fashion by abstracting from existing protocols, standardizing similar interfaces, etc. Since a development always begins with small interfaces, there will be a frequent need to define new 'higher interfaces' that incorporate existing interfaces. ABAP therefore provides a mechanism for defining hierarchy and abstraction in interfaces.

An interface can **contain** other interfaces and also add to these by defining its own attributes, methods and events. The attributes, methods and events specific to this interface form a separate interface that supplements the contained interfaces. An interface that also contains other interfaces as components is also called a **composite interface**. This way arbitrary interface hierarchies can be defined.

Classes that implement a composite interface also have to implement all the interfaces that this composite interface contains.

If a class implements a number of different interfaces, all of which contain another interface, this further interface and its associated attributes exist just once for each instance of the class (see Figure 1).

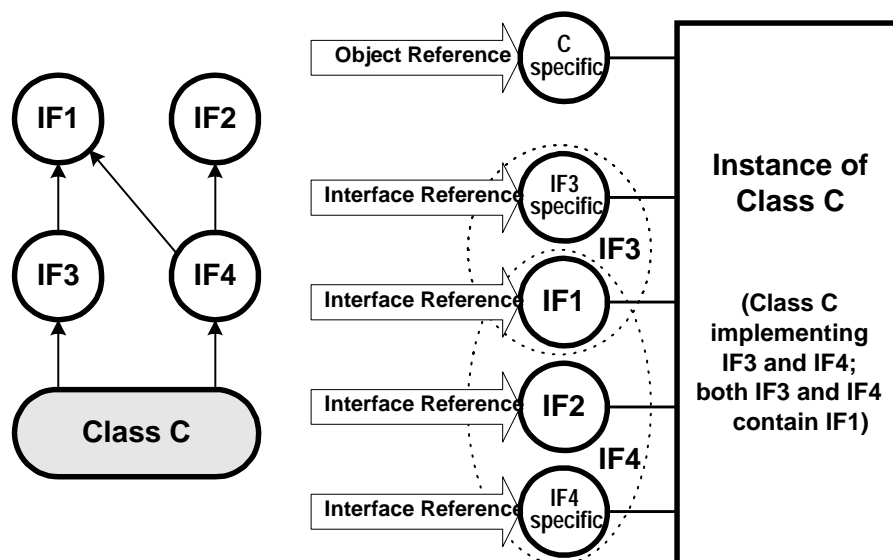


Figure 1: Interfaces that Contain Other Interfaces

Avoiding Naming Conflicts and Aliasing

In order to be able to further develop existing interfaces independently of each other and to define new interfaces independently of others, naming conflicts must be avoided as far as possible. For this purpose, each interface has a separate name space for its components. If interfaces contain other interfaces, the name spaces of the contained interfaces are not part of the name space of the composite interface.

Embedding interface components in the name space of a comprising interface



Through the explicit definition of **aliases** for components of contained interfaces, a composite interface can declare specific components of the contained interface in its own name space.

Extending Interfaces

Supporting Class and Interface Evolution

Since experience shows that the first version of an interface is rarely complete, it should be possible to define new interfaces as *extensions* to existing interfaces. The extended interface thus incorporates the original interface and contains all components defined by that original interface. In ABAP, the extension of interfaces is achieved by making the extending interface contain the original interface and map all its components under the same name in its own name space using Aliases.

A class implementing the new interface just needs to implement the bodies for the new methods. The clients can use the old and new components uniformly through the one new interface. This also allows for a gradual migration of clients and implementers of interfaces.

References to Objects and Interfaces

Reference variables

When an object is created, a reference to the new object is returned to the creator. This can be used for subsequent access to the object. The reference is stored in a **reference variable**. Before the first assignment of a reference to a reference variable, the reference variable contains a **NULL reference** ('INITIAL').

Copying Reference Variables

Object sharing

If the contents of one reference variable are copied into another reference variable, then the referenced object itself is not copied. Instead, the result is that the two reference variables now reference the same object (→ *Object sharing*).

Types of Reference Variables

References to objects are being introduced as a new primitive data type in ABAP. Reference variables are typed.

Object reference variables

When defining an **object reference variable** a class name has to be specified. All objects that can be referenced by these variables belong either to this class or to one of its subclasses.

Interface reference variables

When defining an **interface reference variable** an interface name has to be specified. All objects that can be referenced by these variables must have implemented this kind of interface.

Universal reference variables

ABAP also allows reference variables to be created that can reference arbitrary objects. The contents of any other reference variable can be assigned to them. There is a pre-defined class called OBJECT for these universal reference variables. The OBJECT class can be regarded as an implicit superclass of all other classes.

Since attributes can also be declared as reference variables, objects can of course refer to other objects.

References between objects

Type Checking Assignments

ABAP employs 'mostly static' type checking for objects. When an object reference variable is of type 'reference to class C', then this variable can only hold references to instances of C or a subclass of C. Thus, an assignment of the form "o1 = o2" (where o1,o2 are reference variables) is legal at compile time only when the types of o1 and o2 are compatible, i.e. when o2 has the same or a subclass type of o1. Similarly, you can assign an object reference to an interface reference variable only when the object implements that interface.

There are cases, when it is convenient or necessary to defer type checking to runtime. ABAP has a special statement that allows 'downcast' and general 'type queries' for objects. This way, it is possible, for instance, to ask an object what interfaces it implements.

In general, ABAP follows the rule, that all object references and assignments to them are type-safe. Whenever a type check has to be deferred to runtime, the programmer has to explicitly specify this by special statements or statement variants.

The Object Life Cycle

Creating Objects

Objects are always created both explicitly and dynamically. The class of the object has to be specified either explicitly or implicitly. A reference is returned to the creator, and this can then be used to access the newly created object.

Dynamic creation

If an object with an encapsulated internal state is to be able to guarantee its own consistency, then this must be true from the moment when it is created onward. Therefore ABAP provides a so-called **constructor method** for every class. This method is called by the system automatically when the object is being created and has the task of initializing the object. This method is also known simply as the '**constructor**'.

Using constructors for initialization

There is exactly one constructor for each class. This is a special method⁵ that may be parameterized. Any mandatory parameters have to be supplied when an object is being created.

Uniqueness of the constructor

A subclass cannot override the constructor that it inherits. Instead, the constructor of the subclass must call the constructor of its direct superclass explicitly and supply it with the required parameters. The number and the types of the parameters of the subclass constructor may vary from those of the superclass, however.

Constructors and inheritance

⁵ A class constructor has the same name as the class itself.



Storage Management for Objects

- Garbage collection** Objects are created explicitly, but cannot be deleted explicitly. If a program component no longer requires an object, then it can assign a 'NULL' reference to the appropriate reference variable. This means that the object is no longer referenced by this variable. The runtime system ensures that no object is deleted if there is still a valid reference to the object anywhere in the program. Only when there are no more references to the object the system is free to delete the object at some future point. This procedure ensures that invalid references never occur.
- No destructors** Since the system alone is responsible for deciding when an object should be deleted, there is currently no provision in the language for special methods that are executed immediately before an object is deleted (destructor methods). There is provision however, to automatically release external resources when an object 'dies' or the whole program terminates.

Local and Global Classes and Interfaces

Classes and interfaces can be defined either globally or locally. Global classes and interfaces are stored in the class library, also known as the Object Repository⁶. Local classes can be defined like ordinary types on the top level of programs, reports, module pools, function groups etc.

Summary

- Objects** The concept of an object is central to the object-oriented paradigm. The idea of objects is that the entities involved in a problem should be represented as far as possible one-to-one by objects in the proposed solution. Objects have **attributes** that describe the current state of the object. The behavior of objects is described by their **methods**. A further feature of ABAP objects is that they can both raise and handle **events**.
- Classes** A set of objects having identical interfaces and an identical structure forms a **class**. A class can be defined as a specialization of another class. Subclasses "inherit" all the public attributes of their superclass and also the implementation of the superclass methods. The superclass methods may, however, be overridden in the subclass.
- Interfaces** Objects that have the same interfaces can always be treated in the same way, even if they have been implemented quite differently from each other. ABAP provides the **interface** construct for this purpose. By defining an interface, the common client-visible aspects of two objects can be expressed, even though the objects do not have a common superclass.
- Classes implement interfaces** A class definition can specify that the class implements one or more interfaces. The instances of this class can be used in just the same way as all other

⁶ The contents of the BOR (Business Object Repository, R/3 Release 3.1) as it is today will be integrated into this upcoming object repository.



objects that have the same interfaces. The class itself must define a method body for each of the interfaces' methods.

Interfaces can contain other interfaces. Classes that implement a composite interface must also implement all contained interfaces as well. ABAP also enables interfaces to be extended and thus further developed by adding more attributes, methods and events. This means that existing program components do not have to be modified unnecessarily.

Objects need not be deleted explicitly. Instead, the runtime system provides a garbage collection mechanism.

More information on SAP and object technology can be found on the web at <http://www.sap.com>.

Combining and extending interfaces

Garbage collection