**ABAP History**

**A**BAP Objects, the object-oriented (OO) extension to the ABAP language.

ABAP Objects is a nearly 100 percent upward-compatible extension of ABAP that includes a full set of OO features. We left out some obsolete language elements in the OO context within classes (so it is not truly 100 percent compatible), but you can use objects in all contexts without any restrictions.

We created ABAP Objects because:

•The business object repository (BOR) already presented an OO view of the system to the outside, and we wanted a common programming model for both outside and inside.  
•We wanted the potential benefits of objects within SAP, including better control of complexity, a better means for encapsulation, extensibility, and reuse, and a language basis for patterns and frameworks.  
•We wanted seamless integration with external object models (DCOM and CORBA).  
•We needed objects to be the foundation for a new GUI programming model.  
At the beginning of the ABAP Objects project, we faced a fundamental question: Should we add OO features to the existing ABAP language, invent a completely new language, or integrate an existing OO language into ABAP? We opted to integrate OO features into the existing ABAP language to create ABAP Objects for several reasons:  
•It meant we would immediately inherit all the other features of the ABAP infrastructure such as software logistics, portable database access, TP monitor capabilities, and all database buffering and scalability.   
•Certain features, such as persistence, transactions, events, and GUI frameworks, are hard to add on top of a language. This results in lots of generated code. To support these concepts, you must have your own compiler or interpreter and runtime, ruling out most off-the-shelf languages.  
•Our target was a language for business applications. We wanted it to be simpler than Java and C++ and to leave out complex concepts lacking in benefits.  
•By integrating the *concepts* of OO languages—as opposed to using a specific OO language—into ABAP, we were able to combine the best concepts from these languages and adapt them, when needed, to our specific needs.

Our main design goals were:

•To make ABAP Objects as simple as possible  
•To use only proven OO concepts  
•To comply with external standards whenever possible (for example, to allow mapping to COM, CORBA, and UML)  
•To require much stronger typing (type checking) than traditional ABAP, where type specification is almost always optional.

**The Design Environment**

When designing ABAP Objects, we faced some special ABAP-specific challenges:  
•Extreme variability—ABAP is a meta-application that is massively configurable and extendable by five layers.   
•ABAP is an interpretive system, so there is never a shutdown-change-recompile-restart cycle.   
•Software development logistics allow hundreds of developers to work on the same system parts, and logistics must not be hindered by the language. The change process must be considered and supported by certain language features (mostly resulting in some deliberate deviations from the general strong typing policy). You cannot assume that developers can always perform synchronous global changes in the system.

Some of ABAP Objects’ key design elements include:

•Classes are not structures with methods (as in C++) but a new kind of type.   
•Reference semantics for objects. ABAP is completely value-based. There are no references in the language, and a MOVE always copies its values (even with field-symbols). Objects, on the other hand, can only be accessed by references (there are no embedded objects). This results in a clear separation of values and references. An object reference always has reference semantics. This is the only place in ABAP where objects are shared and never copied. Values (the traditional data) are always copied.  
•Single inheritance with a separate concept of interfaces. Here we follow a path similar to Java. Most multiple inheritance applications are actually interface cases, and the few extra capabilities of multiple inheritance definitely don’t warrant the complexity this introduces into the language.  
•ABAP Objects integrates complex concepts, most notably events and dynamic method invocation. Events are introduced similarly to how they exist in Visual Basic. They are outbound methods that can have parameters. This is a natural model for GUI applications. We also separated the event registration into a separate statement (SET HANDLER), which gives you the flexibility to define all kinds of topologies of sender-to-handler connections (1:1, 1:n, n:1, n:m).   
•Of course, ABAP Objects has its own storage management (garbage collection) and a suite of tools such as the Class Builder, Class Browser, and debugger support.

**ABAP Objects Basics**

ABAP Objects is based on classes. Classes are pieces of program code that describe objects by defining their components. Typical object components are attributes (data), which describe an object’s state, and methods (functions), which describe an object’s behavior. Objects are instances of classes. Many objects can be created from one class, but each object has its own attributes.

ABAP Objects provides a set of new statements that defines classes and handles objects. These statements can be used in any kind of ABAP program.

Classes, and therefore objects, consist of at least two layers—an inner and an outer layer. The externally visible layer of an object is made up of public components. Public components can be accessed directly by all users and form an object’s external point of contact. Private components are only visible within objects of the same class. The public section usually contains few attributes. The state of an object is generally kept in private attributes and manipulated by public methods. This way an object can ensure its own consistency.

Outside ABAP Objects, the only instances in ABAP are instances of whole ABAP programs such as reports, module pools, or function groups. Each time a program is executed, a program instance is implicitly created, which then runs in a separate ABAP process. Calling an external procedure (a subroutine or a function module) also creates an ABAP program instance, which contains the procedure. However, repeated calls always use the same one instance. Such program instances live as long as the application program is running and cannot be handled explicitly.

With ABAP Objects, it is now possible to explicitly handle instances. The instances of classes, namely objects, are created explicitly with ABAP statements. They have a unique identity (independent of their attribute values) and are addressed by references. They live only as long as they are needed.

**Classes.** Classes are templates for objects. You can either define a class locally in an ABAP program, or globally in the class library using the Class Builder tool in the ABAP Workbench. The class library is part of the ABAP repository. Class library classes are stored in special programs called class pools. A class pool is automatically generated for each global class that you create with the Class Builder. The Class Builder also generates the coding frame that defines the classes in the class pool. Global classes are visible to all ABAP programs of the ABAP System.

To define local classes in an ABAP program, such as a report or a function group, you must type the corresponding statements manually into the program. A local class is visible only inside the program in which it is defined.

Classes are defined between the CLASS and ENDCLASS statements. A class definition consists of a declaration part, in which the components are defined, and an implementation part, in which the methods are implemented.

**Component visibility.** Each component must be declared in one of three possible visibility sections. All components declared in the PUBLIC SECTION can be accessed by all clients of the class. All components declared in the PROTECTED SECTION can be accessed in the methods of the class and its subclasses (see inheritance). All components declared in the PRIVATE SECTION can only be accessed in the methods of the class. The public section forms the outer layer, or external interface, of the class. The protected and the private sections form the inner layer. Their components cannot be accessed by external users.

**Class components.** Possible class components are attributes, methods and events. Events will be explained later in this article.

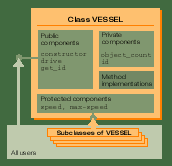
•**Attributes:** are the internal data variables within a class. They can have any ABAP data type. We distinguish between instance attributes and static attributes. Instance attributes are declared by DATA and determine the state of an instance. You cannot work with instance attributes without creating an object first. Static attributes are declared by CLASS-DATA and determine the state of a class, which in a way applies to all instances. Static attributes form a data set that is shared by the whole class and all of its objects. You do not need to create an object to work with static attributes.

**•Methods:** are the class’ procedures. They can access all class attributes and can therefore change the state of an object. They have a parameter interface similar to the interface of function modules. They can have named IMPORTING, EXPORTING, and CHANGING parameters, which can be optional or required and can be passed either by reference or by value. As with attributes, we distinguish between instance methods and static methods. Instance methods are declared by METHODS and can access all the attributes of a class. Static methods are declared by CLASS-METHODS and can only access the static attributes of a class.

Listing 1, defines a class vessel. The components of class vessel are the attributes speed, max\_speed, object\_count, and id, and the methods constructor, drive, and get\_id.

|  |
| --- |
| CLASS vessel DEFINITION.  PUBLIC SECTION.  METHODS:  constructor,  drive IMPORTING speed\_up TYPE i,  get\_id RETURNING value(id) TYPE i.  PROTECTED SECTION.  DATA: speed TYPE i,  max\_speed TYPE i VALUE 100.  PRIVATE SECTION.  CLASS-DATA object\_count TYPE i.  DATA id TYPE i.  ENDCLASS.  CLASS vessel IMPLEMENTATION.  METHOD constructor.  object\_count = object\_count + 1.  id = object\_count.  ENDMETHOD.  METHOD drive.  speed = speed + speed\_up.  IF speed > max\_speed.  speed = max\_speed.  ENDIF.  ENDMETHOD.  METHOD get\_id.  id = me->id.  ENDMETHOD.  ENDCLASS. |
| **Listing 1. *Statements for defining a class.*** |

Figure 1, shows how the class vessel is composed by its visibility sections.



**Figure 1 *Visibility sections of a class.***

**Method implementation.** Each method declared in the declaration part of a class must be implemented in its implementation part. Each implementation is enclosed between the METHOD and ENDMETHOD statements. You can use almost all ABAP statements available in subroutines or function modules to implement the method, but ABAP Objects does use a stricter syntax in some places to avoid obsolete features. Methods may contain declarations of local data and data type.

**Constructors.** Implicitly, each class has an instance constructor method with the reserved name constructor and a static constructor method with the reserved name class\_constructor. The instance constructor is executed each time you create an object (instance) with the CREATE OBJECT statement, while the class constructor is executed exactly once before you first access a class. The constructors are always present. However, to implement a constructor you must declare it explicitly with the METHODS or CLASS-METHODS statements. An instance constructor can have IMPORTING parameters and exceptions. You must pass all non-optional parameters when creating an object. Static constructors have no parameters.

**Objects and object references.** Objects are instances of classes. From the user’s point of view, there is almost no difference between creating objects based on global classes and creating them on local classes. The statements for creating and handling objects are exactly the same for both types of classes. Visibility distinguishes between global and local classes. While any program can create objects from classes in the class library, only the defining program can create objects from its own local classes.

**Reference variables.** Objects live in the internal session of an ABAP program and are accessed only via reference variables. Object reference variables contain references (pointers) to objects and must be declared before creating an object and accessing its components.

To declare a reference variable, you use the new data type REF TO class. Like other variables, reference variables can be components of structures or internal tables. You initialize a reference variable with the CLEAR statement. The initial value of a reference variable is a reference that does not point to an object.

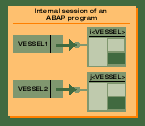
**Instances.** Once you have declared a reference variable oref with the type of a class, you can create a class instance. Use the statement CREATE OBJECT oref, which creates an instance of the class with the type of oref and places a reference to the new instance into oref. Each CREATE OBJECT statement creates a new object.

You can create any number of instances of the same class in a program. Those instances are fully independent of each other, and they have their own identity and attributes within the program. You can assign references between reference variables using the MOVE statement or pass them as parameters. ABAP Objects’ reference variables are always type-safe in the sense that they can only contain pointers to objects that conform to the interface given by the reference variable’s declaration type. If that check is not possible statically, you can cast reference variables at run time.

Listing 2 declares two reference variables, vessel1 and vessel2, and creates two instances of class vessel with the two reference variables pointing to them. Methods of the objects are then called.

|  |
| --- |
| DATA: vessel1 TYPE REF TO vessel,  vessel2 TYPE REF TO vessel.  DATA vessel\_id TYPE i.  CREATE OBJECT: vessel1, vessel2.  CALL METHOD: vessel1->drive( 50 ),  vessel2->drive( 80 ).  vessel\_id = vessel1->get\_id( ).  WRITE: / 'Vessel ID is', vessel\_id.  vessel\_id = vessel2->get\_id( ).  WRITE: / 'Vessel ID is', vessel\_id. |
| **Listing 2. *Using objects and their components.*** |

Figure 2 shows the two reference variables, each pointing to an instance of class vessel. The names of the objects are shown in the same format (id<classname>) as the contents of reference variables in the ABAP Debugger. These names cannot be used directly in the program.



**Figure 2 *Reference variables and objects.***

**Garbage collection.** An object is in use as long as there is at least one reference pointing to it. When there are no more references pointing to an object, the memory space occupied by that object can be released. ABAP Objects provides a garbage collector, which ensures that the memory is collected automatically.

**Accessing the components of objects.** To access an instance component from outside the defining class, you use the object component selector ->, which combines the object (reference) with the component name. Similarly, the class component selector => combines the class with the component name when you access a static component.

To access attributes, you write oref->attr for instance attributes and class=>attr for static attributes. To call a method, you write CALL METHOD oref->meth for an instance method and CALL METHOD class=>meth for a static method. CALL METHOD has almost the same syntax as CALL FUNCTION, but ABAP Objects adds functional methods. These are characterized by the fact that besides IMPORTING parameters, they have exactly one RETURNING parameter. This is an advantage because they can be used like an ordinary variable directly in certain expressions. Instead of entering a variable name, you simply write oref->meth(...) or class=>meth(...) specifying the required IMPORTING parameters within the parentheses. During the execution of the expression the method is evaluated, and the value of the RETURNING parameter is used in the expression.

Inside methods, components of the same class can be addressed directly by their names, where the compiler prepends the self reference ME implicitly. Only in special cases—for example, when an attribute is hidden by a local data object or parameter with the same name—are you required to use just the ME->attr notation to explicitly select the attribute (see an example in the constructor in Listing 3). ME can also be used when a method wants to pass a reference that points to its own object.

|  |
| --- |
| CLASS ship DEFINITION  INHERITING FROM vessel.  PUBLIC SECTION.  DATA name TYPE string READ-ONLY.  METHODS: constructor  IMPORTING name TYPE string,  write,  drive REDEFINITION.  ENDCLASS.  CLASS ship IMPLEMENTATION.  METHOD constructor.  CALL METHOD super->constructor.  max\_speed = 30.  me->name = name.  ENDMETHOD.  METHOD write.  DATA id.  id = me->get\_id( ).  WRITE: / name,  'is vessel', id,  'and has speed', speed.  ENDMETHOD.  METHOD drive.  speed = speed + speed\_up.  IF speed > max\_speed.  max\_speed = 0.  speed = 0.  ENDIF.  ENDMETHOD.  ENDCLASS. |
| **Listing 3. *Statements for deriving a subclass.*** |

**Inheritance.** Inheritance allows you to derive new classes from an existing one. The new class inherits all components of the existing class and is known as the subclass. The existing class is known as the superclass. A subclass reuses the code of its superclass, but only the public and protected components of the superclass are visible in its subclasses. You can define additional components in the subclass, making it more specialized than the superclass. A subclass itself can become a superclass of additional new classes. This allows you to introduce several degrees of specialization.

**Subclasses.** To define a subclass, you use the INHERITING FROM clause in the CLASS statement of the declaration part. ABAP Objects supports single inheritance. Classes can have several direct subclasses, but only a single superclass. A special kind of multiple inheritance in ABAP Objects is provided by interfaces (which we will get to shortly).

When a subclass inherits from a superclass that is itself a subclass of another class, the classes involved form an inheritance tree, which becomes more specialized moving from top to bottom. The components of a superclass are passed on to all subclasses within the tree. The implicit root node of all ABAP Objects inheritance trees is the predefined class OBJECT.

In Listing 3, a subclass ship is derived from superclass vessel. The subclass ship specializes its superclass by adding a new attribute name and a new method write. Furthermore, class ship has its own constructor and redefines the method drive.

**Redefinition.** In subclasses, you can make the public and protected instance methods of all preceding superclasses specialized by redefining them using the REDEFINITION clause of the METHODS statement. You cannot change the interface of a redefined method. Redefined methods are merely implemented differently under the same name. Within a redefined method, you can use the pseudoreference SUPER to access the method with the same name in the superclass. It is always a good idea to call the superclass method that you are redefining before specializing it.

If you define an instance constructor in a subclass, you must call the instance constructor of the superclass by using CALL METHOD super->constructor in its implementation. Before the call, the subclass constructor behaves like a static method. It can access only static attributes in that part of the constructor. After the call, it behaves like an instance method and can also access instance attributes. This is to ensure that the superclass is completely initialized before the subclass accesses its attributes.

**Abstract and final.** ABAP Objects also provides ABSTRACT and FINAL clauses like other OO languages. With the ABSTRACT and FINAL clauses for the METHODS and CLASS statements, you can define abstract and final methods or classes. When a method is declared abstract, it doesn’t have an implementation in this class, but it expects a subclass to provide its (specialized) implementation. When a class has one or more abstract methods, the class as a whole becomes abstract or incomplete. Therefore, you cannot create abstract class objects. The FINAL clause allows you to restrict inheritance. Final methods cannot be redefined in a subclass, and final classes cannot have any further subclasses.

**Polymorphism.** Polymorphism is a very important OO concept. Inheritance provides one means of polymorphism. Its effect is best seen when we consider using objects via a reference variable. Polymorphism means that a reference variable with the type REF TO vessel can always contain references to instances of any subclass of that class vessel, such as to an instance of ship or boat. Since a subclass has at least all the components of all its superclasses, you can access all the components of vessel in any subclass. By using a higher level reference, you simply ignore the components added in subclasses.

However, the most important part of polymorphism in terms of inheritance is the methods. Consider the example above: A class ship is inherited from class vessel and has redefined the method drive. When we now have a pointer to an instance of ship in a reference variable of type REF TO vessel, and we call the method drive, the drive method of the class ship is executed, even though the reference to that instance is of type REF TO vessel.

The general rule is: When calling a method, the most specialized redefinition (in a subclass) gets called. This method can then directly call its superclass implementation. Thus, with inheritance, a method call goes as far down as possible, and your own code must call the implementations of this method upwards. In ABAP Objects, you are only allowed to call a method’s direct predecessor implementation. You can only go one level up with CALL METHOD SUPER->m, and you can only call your own method and not other methods of your superclass. (These restrictions are not for technical reasons but to guide the users of ABAP Objects towards proper use of inheritance.)

Reference variables can always hold reference to subclass instances. In particular, the root class OBJECT can hold references to any object.

Using the normal method call or attribute access notation, you can only access components of the class that was used to define the reference variable. Only a dynamic invoke allows the user to access statically unknown methods. For example, if a subclass ship had an additional method sink, you could not call it statically through a reference of type REF TO vessel, but you could call it using the dynamic method call.

Listing 4 declares two reference variables, one of type vessel and one of type ship. A class ship object is created. The IMPORT parameter of its constructor is filled. The object can be accessed by both reference variables. In both cases, the method drive that is declared in class vessel is called. But you cannot call the method write with reference variable vessel.

|  |
| --- |
| DATA: vessel TYPE REF TO vessel,  ship TYPE REF TO ship.  CREATE OBJECT ship EXPORTING name = 'Titanic'.  CALL METHOD ship->drive( 20 ).  MOVE ship TO vessel.  CALL METHOD vessel->drive( 10 ).  CALL METHOD ship->write. |
| **Listing 4. *Using an objects of a subclass.*** |

The implicit calling of methods that are redefined in subclasses means that, with the superclass’s interface, a method call accesses different method implementations in different subclasses. This has an important consequence: A subclass should never change the semantics of a method via redefinition. A subclass must always do more than the superclass but never take anything away.

**Interfaces.** The public components of a class define the external point of contact for the class. They define the semantic interface specific to this class. As an illustration, a Customer class would have a CustomerId, and an Order class would have an OrderId as well as lots of specific methods.

However, there are many cases when we want another kind of polymorphism, one that is independent of inheritance. Consider the following examples:

The ArchiveManager class wants to define what it needs in order to archive a class in a special interface and then access all different classes uniformly through that interface. It doesn’t care what specific class it is accessing. It only cares about the archiving aspect. All classes that want to be archived must implement the IF\_Archive interface. In the same way, a WorkflowManager class would define an IF\_Workflow interface, and all classes that want to participate in the workflow must implement this interface.

The general concept of interfaces is this: An interface is a separate language construct that is allowed to define components (such as in the public section of a class) but without implementation. Classes can implement one or more interfaces in their public section, thereby extending this section.

Interfaces can also be used to define reference variables of type REF TO interface. These interface references only show those components that are defined in the interface, regardless of the implementing class. When a class implements an interface, we can get a reference to any implemented interface by way of simple assignment of the form iref = oref.

**Interfaces in ABAP Objects.** Like classes, you can define interfaces globally in the class library or locally within an ABAP program. Interfaces are defined between the INTERFACE and ENDINTERFACE statements. They can contain the same components as classes. An interface’s methods are not implemented in the interface itself, but must be implemented in the classes that implement the interface. You do not have to assign the interface components to a visibility area. Instead, all components automatically belong to the public section of the class implementing the interface.

You implement an interface in a class by using the statement INTERFACES in the public section, which causes the components of the interface to be added to the public components of that class, all with names of the form intf~comp. All components of an implemented interface appear within the class with the name prefix intf~. To access a component comp of an interface within a class or when using a class reference, you must use the name intf~comp. However, when using an interface reference, you can specify the interface components directly, since the interface reference only shows the components of that interface. While we use the names intf~comp inside the implementing class, the users of the interface just write iref->comp.

Implemented interface components are fully equivalent to normal class components. They extend the outer layer (public section) of the class. All users can access both the class-specific public components as well as the components of the interface.

One class can implement several interfaces, and each interface can be implemented by many classes. When a class implements an interface, it immediately has its attributes and events (see Listing 5), but it must implement all methods of the implemented interfaces in its implementation part. The interface’s methods can be implemented differently in each implementing class. Therefore, in addition to inheritance, interfaces provide another means of polymorphism in ABAP Objects. As for redefined methods with inheritance, the implementation of an interface method should confirm to the semantics defined in the interface.

|  |
| --- |
| INTERFACE status.  METHODS write.  ENDINTERFACE.  CLASS ship DEFINITION  INHERITING FROM vessel.  PUBLIC SECTION.  INTERFACES status.  ...  ENDCLASS.  CLASS ship IMPLEMENTATION.  ...  METHOD status~write.  DATA id.  id = me->get\_id( ).  WRITE: / name,  'is vessel', id,  'and has speed', speed.  ENDMETHOD.  ...  ENDCLASS.  CLASS coast\_guard DEFINITION.  PUBLIC SECTION.  INTERFACES status.  ALIASES write FOR status~write.  PRIVATE SECTION.  DATA caller TYPE string.  ENDCLASS.  CLASS coast\_guard IMPLEMENTATION.  METHOD status~write.  IF caller IS INITIAL.  WRITE:  / 'Coast guard received no call'.  ELSE.  WRITE:  / 'Coast guard received a call from',  caller.  ENDIF.  ENDMETHOD.  ENDCLASS. |
| **Listing 5. *Defining and implementing interfaces.*** |

Listing 5 defines an interface status. It contains one method, write. The class ship replaces its own method write by implementing the interface status. Another class, coast\_guard, is defined that also implements status. Both classes implement the interface method status~write.

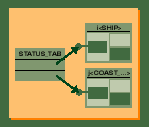
**•Nested interfaces:** By using the INTERFACES statement within the interface definition, you can define nested interfaces. When an interface contains a nested component interface, this inner interface appears as one component. The components of the contained interface are not automatically visible in the enclosing interface. To make a component of a contained interface component visible on the higher (enclosing) level, you can use the ALIASES statement.

**•Interface references:** Besides reference variables declared with respect to a class, ABAP Objects also provides reference variables declared with respect to an interface. Those interface reference variables expose only the components defined in the interface, regardless of the implementing class.

Listing 6 declares a variable status and an internal table status\_tab as interface reference variables. The contents of two class reference variables, pointing to objects of ship and coast\_guard, are appended to that internal table. By looping on the internal table, the interface method write is accessed in both objects. Listing 6 shows how objects of two different classes are accessed by reference variables of one kind. Although ships and coast guards do not have much else in common, the user can expect the same semantics from the interface method write.

|  |
| --- |
| DATA: status TYPE REF TO status,  status\_tab TYPE TABLE OF REF TO status.  DATA ship TYPE REF TO ship.  DATA station TYPE REF TO coast\_guard.  CREATE OBJECT ship  EXPORTING name = 'Titanic'.  APPEND ship TO status\_tab.  CREATE OBJECT station.  APPEND station TO status\_tab.  LOOP AT status\_tab INTO status.  CALL METHOD status->write.  ENDLOOP. |
| **Listing 6. *Using Interfaces.*** |

Figure 3 shows the internal table of interface references pointing to objects of different classes that implement a common interface.



**Figure 3 *Using interface references.***

**Events.** Events generally announce that an object’s state has changed or that something happened that may be of interest to others. They are like outbound method calls and enable a different kind of coupling between objects. They are especially useful in a GUI environment, but are also good for loosely coupling subsystems and for workflow.

The basic concept of methods is: When one object A calls a method of another object B, A must know B. A must have an attribute that is a reference to B.

With events, the situation is different: They work by publish and subscribe. One class A defines an event e that may be of interest to other classes/objects. When class B wants to react to the event e of A, it defines a handler method h for that event. At run time, an object of B registers its method h as a handler for that event of a sender object of class A. When the object A now raises event e, all registered handlers for that event are notified or called. The key issue is that a handler can be attached to do more without changing the sender’s code.

**•Events in ABAP Objects:** Events are another kind of component in classes. They are declared in the declaration part of a class and raised in its methods. You declare events by using the statements EVENTS for instance events and CLASS-EVENTS for static events. Events can have EXPORTING parameters to pass parameters from the triggering object to the handler. Each instance event has an additional implicit parameter called SENDER. The SENDER data type is a reference variable to the respective sender class.

Static events can be raised in any method of the declaring class, whereas instance events can only be raised in instance methods. You raise an event in a method using the RAISE EVENT statement, and you pass arguments using the EXPORTING clause. For instance events, the system automatically passes the implicit parameter SENDER,which is a reference pointing to the object raising the event.

Any class may contain event handler methods for events of other classes. The event handler method signature (names and types of parameters) is not repeated but is taken from the event to be handled. However, the event handler method does not have to accept and use all of the parameters that the event defines. An event handler method can accept the implicit parameter SENDER like any other IMPORTING parameter, allowing it to access the triggering instance. The declaration of events and respective event handler methods in classes constitutes the static part of publish and subscribe.

Listing 7 declares an event emergency\_call in class ship. If the attribute speed becomes higher than max\_speed in method drive, the event is raised. A method receive is declared as an event handler in class coast\_guard.

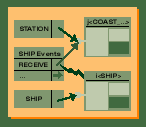
|  |
| --- |
| CLASS ship DEFINITION  INHERITING FROM vessel.  PUBLIC SECTION.  ...  EVENTS emergency\_call.  ...  ENDCLASS.  CLASS ship IMPLEMENTATION.  ...  METHOD drive.  ...  IF speed > max\_speed.  ...  RAISE EVENT emergency\_call.  ENDIF.  ENDMETHOD.  ...  ENDCLASS.  CLASS coast\_guard DEFINITION.  PUBLIC SECTION.  ...  METHODS receive  FOR EVENT emergency\_call  OF ship IMPORTING sender.  ...  ENDCLASS.  CLASS coast\_guard IMPLEMENTATION.  ...  METHOD receive.  caller = sender->name.  CALL METHOD write.  ENDMETHOD.  ...  ENDCLASS. |
| **Listing 7. *Declaring raising, and handling events.*** |

**•Handling events:** If you declare an event handler method in a class, the instances of the class, or the class itself, are then capable of handling the event. If you want an event handler method to react to an event, you must define at run time the trigger to which you want it to respond with the SET HANDLER statement. It links a list of handler methods to corresponding event triggers—either instance events or static events. The SET HANDLER statement also has an addition ACTIVATION, which allows you to register and deregister handlers dynamically. Triggering events in methods and registering event handler methods at run time constitutes the dynamic part of publish and subscribe.

Listing 8 registers the method receive in an instance of class coast\_guard as a handler for the event emergency\_call in an instance of class ship.

|  |
| --- |
| DATA: ship TYPE REF TO ship,  station TYPE REF TO coast\_guard.  CREATE OBJECT: ship  EXPORTING name = 'Titanic'.  station.  SET HANDLER station->receive FOR ship.  DO 5 TIMES.  CALL METHOD ship->drive( 10 ).  ENNDO. |
| **Listing 8. *Registering handler methods for events.*** |

For each SET HANDLER statement, the system creates an entry in an event handler table (see Figure 4), where a handler table belongs to each instance or class that can trigger instance or static events. It is invisible to the user and contains the names of the handler methods and references to the registered handling instances. When an event is triggered, the system searches the event handler table of the corresponding object or class and executes the listed methods in their instances in the case of instance handler methods, or classes in the case of static handler methods. You can delete single entries from the handler table using the ACTIVATION addition of the SET HANDLER statement. If the triggering instance is deleted by the garbage collector, the entire handler table, including all its references, is deleted.



**Figure 4 *Event handler table of an object.***

A reference to an instance in a handler table counts as a use of the instance, just as a reference in a reference variable does. Handler table entries, therefore, affect the lifetime of objects. This means that in our example, the instance j<coast\_guard> is not deleted by the garbage collector as long as it is registered for event handling, even if the reference variable station is cleared.

Finally, events, as well as their handler methods, can also be defined in interfaces. Additionally, there are some powerful mass registration variants to register a handler for all instances of a class and all classes implementing a certain interface.

***Jùrgen Heymann*** *is senior developer in the SAP ABAP Language Group. He is a key architect in the ABAP Objects project. You can reach him at juergen.heymann@sap-ag.de.*

**Horst Keller** is senior technical writer in the SAP ABAP & GUI Group. He documents the ABAP language with an emphasis on ABAP Objects. He also develops and teaches classes on ABAP programming. You can reach him at [*horst.keller@sap-ag.de.*](mailto:horst.keller@sap-ag.de)