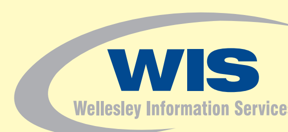
Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Developer Should Give It a Second Look

Horst Keller and Gerd Kluger

This article originally appeared in the September/October 2004 issue of SAP Professional Journal and appears here with the permission of the publisher, Wellesley Information Services. For information about SAP Professional Journal and other WIS publications, visit **www.WISpubs.com**.



Horst Keller, NetWeaver Development Tools ABAP, SAP AG

Gerd Kluger, NetWeaver Development Tools ABAP, SAP AG

(complete bios appear on page 53)

It is a puzzling phenomenon. SAP introduced ABAP Objects, the object-oriented (OO) extension to the ABAP programming language, some time ago as part of SAP Basis Release 4.6. ABAP developers, in turn, gained the ability to tap into the myriad merits of OO programming, which significantly improves productivity by speeding development time and increasing application maintainability. Yet many ABAP developers still cling to the long-standing procedural approach to ABAP development — because that's the way they've always done it, it works, and they see no reason to change — leaving the OO capabilities of ABAP Objects untapped. This article aims to change that by showing you just what you have to gain, and how to make the leap.

Most of the existing articles on OO programming in ABAP are rather abstract and use simplified examples that have no relevance to actual business programming. By using an example that is both simple *and* practical to compare and contrast procedural ABAP programming and programming with ABAP Objects, this article clearly demonstrates the advantages you can gain from using ABAP Objects[1]:

1. ABAP Objects establishes an advanced level of data encapsulation that improves the maintainability and stability of your programs. In procedural programming, a program's global data area can encompass a large number of disparate data objects that end up mixed together in a single memory area, where one function might change the application state in a way other functions do not expect. Keeping the state stable thus requires a lot of discipline and

[1] This article applies to SAP Release 4.6 and higher. Although a small portion of syntax in the code samples is available only in Release 6.10 and higher, you should still be able to easily follow along if you are running 4.6.

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

maintenance effort. In OO programming, the state is kept in "objects" that separate externally and internally visible data and implementations, ensuring that functions ("methods" in the OO idiom) work only with the data they should, attributes defining an object's state remain intact, and applications using that object continue to work correctly.

2. ABAP Objects provides you with the ability to instantiate multiple instances of a single class ("objects" containing data and the functions that operate on that data), each with its own variation of the characteristics defined in the class, and to use an object's own functions to change its state. Together with the ABAP Objects automatic garbage collector, this means that developers do not have to deal with manually associating data and functionality for each and every object, or explicitly controlling the lifetime of objects. In procedural programming, which does not support multiple instantiation, data and functions are separate. You work with stateless functions that have to be initialized via their parameter interface with every call, and then manually cleared from memory.

3. ABAP Objects greatly enhances code reuse through "inheritance," which is a key property of the OO programming paradigm. With inheritance you can derive some or all of the characteristics from a general procedure ("method" in the OO idiom) into various specialized procedures simply by programming the difference between the two, which results in smaller procedures and, in turn, improved maintainability. In procedural programming, you often suffer from the "all-or-nothing" limitation: you must either call existing procedures (subroutines or function modules) as they are or implement entirely new ones.

4. ABAP Objects enables you to work with an object's business logic through a standalone interface, rather than working with it directly, which relieves you from having to know exactly how a certain functionality is implemented. It also enables you to later alter an implementation without modifying the coding that uses the interface, so that modifications of standard code become much less problematic.[2] Procedural programming does not support the concept of standalone interfaces as a point of contact to an object — the interface concept is basically restricted to parameter interfaces of procedures (subroutines or function modules).

5. ABAP Objects makes it easy to incorporate event-driven programming models. Applications can be loosely coupled through a "publish and subscribe" model, where a caller (the event) is not statically bound to the called procedure (the event handler, or "callee"). This provides greater flexibility than the procedural approach, where the coupling is much tighter and the flow of program control is more predetermined.

Granted, most (but not all!) of the key benefits of ABAP Objects require an OO approach to application development. While remodeling your entire development approach may seem daunting or just plain overkill, we assure you it's not, and believe you will reach the same conclusion by the time you've finished reading this article. Of course, no one is suggesting you remodel your running applications — we simply recommend that you begin leveraging ABAP Objects for your upcoming projects.

And even if you remain steadfast in the conviction that the procedural approach to ABAP development works just fine for you — along the lines of "if it's not broken, don't fix it" — keep reading. You can still improve your ABAP programs *without* fully embracing the OO programming model simply by using ABAP Objects classes and methods instead of function modules and subroutines:

1. ABAP Objects is more explicit, and therefore simpler to use. For example, you are less bound to the ABAP runtime's implicit control of program flow when working with ABAP Objects. This frees you to define the behavior of your

[2] Business Add-Ins (BAdIs), which are the successors to function exits, use interfaces to enable customers to enhance SAP standard code.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

ABAP Objects at a Glance

Chronology

1. **SAP Basis Release 4.5** delivered the first version of ABAP Objects by extending the ABAP programming language with classes, interfaces, and the ability to create objects from classes (i.e., class "instances").

2. **SAP Basis Release 4.6** delivered the complete version of ABAP Objects by enabling "inheritance," which is the key feature of object-oriented (OO) programming for many programmers, and allowing the creation of a composite interface from several component interfaces.

3. **SAP Web Application Server 6.10/6.20**, the successor to SAP Basis, enhanced ABAP Objects with features like "friendship" between classes and the introduction of Object Services for storing objects in the database.*

4. **SAP Web Application Server ABAP 6.40**** introduced the "Shared Objects" concept, which allows objects to be stored in the shared memory of an application server, so that they can be accessed by all programs on that server.***

---

* For more on Object Services, see the article "Write Smarter ABAP Programs with Less Effort: Manage Persistent Objects and Transactions with Object Services" in the January/February 2002 issue of *SAP Professional Journal*.

** SAP Web Application Server 6.40 is the latest ABAP release and is part of the SAP NetWeaver '04 platform. Unlike SAP Web Application Server 6.10/6.20, which supports only ABAP, SAP Web Application Server 6.40 supports both the ABAP and Java/J2EE runtimes. This article deals solely with the ABAP portion of SAP Web Application Server.

*** Watch for an upcoming *SAP Professional Journal* article on Shared Objects.

(continued on next page)

program yourself, instead of having to understand and obey an external control that is governed by the procedural (i.e., report and dialog) programming model.

2. ABAP Objects offers cleaner syntax and semantic rules. For example, obsolete and error-prone language concepts are explicitly forbidden inside ABAP Objects classes. In contrast, you can still use them in procedural ABAP, in which case the syntax check presents you with a warning in some critical cases, nothing more.

3. ABAP Objects is the only way you can use new ABAP technology. For example, all new GUI concepts, such as the SAP Control Framework (CFW) and Business Server Pages (BSPs), are encapsulated in ABAP Objects classes. With procedural ABAP, you are stuck with classical screen (Dynpro) and list processing.

So even if you have no immediate plans to implement a full-fledged OO programming style for future application development, you can still use ABAP Objects to improve the quality of your ABAP code, making it less error-prone and easier to maintain. In the latter portions of this article, we will show you how.

So how exactly do procedural ABAP and ABAP Objects stack up against one another? We'll answer this question in detail over the following sections. (For a brief background on ABAP Objects before diving into the comparison, see the sidebar above.)

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

(continued from previous page)

Key Facts

☑ ABAP Objects is an extension of the ABAP programming language.

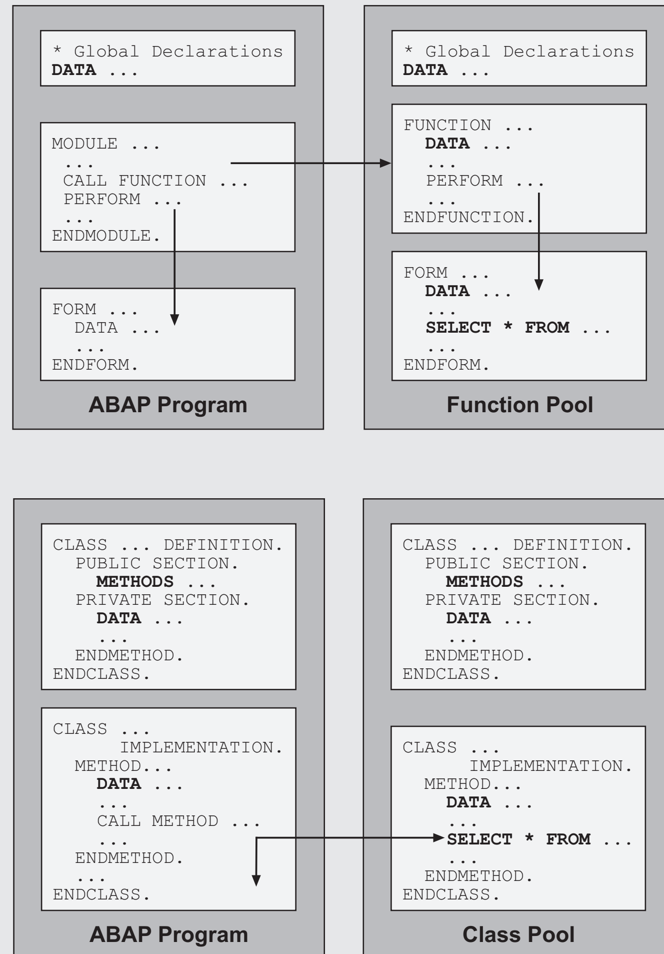
☑ ABAP Objects is designed to be downward-compatible.

☑ SAP devised ABAP Objects to help programmers maximize code reuse.

☑ Since the shipment of ABAP Objects, the ABAP runtime environment has supported both the procedural and OO programming models:

- In procedural programming (see the diagram to the upper right), execution always starts in a dialog module or an event block like START-OF-SELECTION, for example. You work with your program's global data inside these processing blocks. You can modularize your programs internally with **subroutines** or externally by calling **function modules** as external procedures. In addition to accessing the global data of your program, these procedures can contain temporary local "helper" variables that assist with the procedure's internal activities.

- In OO programming (see the diagram to the lower right), the



✓ *Note!*

This article is based in part on a presentation given at SAP TechEd '03. For more information, visit ***www.sapteched.com***.

Five Reasons OO Programming Is Better Than Procedural Programming

In the following sections, we outline the key benefits of OO programming. While these benefits are common to most OO languages (C++, Java, etc.), here the

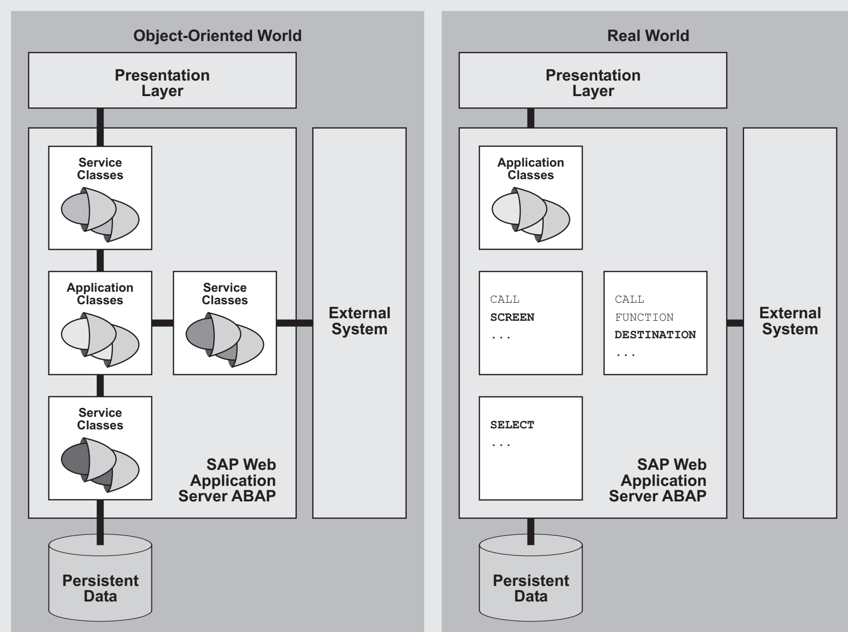
 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

only structuring units are **classes**. Instead of working with global data, you work with "objects," which are instances of classes. These objects encapsulate the state and behavior of self-contained parts of the application. The application state is represented by "attributes" that supersede the global variables used in the procedural approach. The application behavior is determined by methods that modify the object's attributes or call methods of other objects.

☑ The ABAP runtime support for both the procedural and ABAP Objects programming models enables you to use ABAP Objects classes in any existing ABAP program — in executable programs, module pools, or function pools, for example. This enables you to use new techniques based on ABAP Objects, such as new user interfaces, for example, without the need to completely rewrite your existing code.

☑ Currently, most coding that involves ABAP Objects is a mixture of procedural ABAP coding and pure OO coding, as shown in the diagram below.

The left side of the diagram depicts a pure OO world, where all code is written in ABAP Objects and wrapped in application classes. You do not directly access the presentation layer (e.g., SAPGUI or Business Server Pages), persistent data (e.g., database tables or other files), or external systems from inside application classes. Instead, access to external layers is wrapped in service classes, which are taken from the class library. For example, for access to the presentation layer, the following class-based frameworks are available:



(continued on next page)

✓ Note!

For demonstration purposes, the example class definitions are shown as you would see local classes in the ABAP Editor. When working with global classes, you would only see the method implementation coding as plain ABAP text; all other properties (the declaration parts of the classes) are maintained graphically in the Class Builder.

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

(continued from previous page)

the SAP Control Framework (CFW), Desktop Office Integration (DOI), and Business Sever Pages (BSPs). For access to the database layer, you can use Object Services classes and interfaces, which are available with SAP Web Application Server 6.10 and higher.

Although a pure OO world is technically possible, most real-world implementations use a mixture of procedural ABAP and ABAP Objects, as shown on the right side of the diagram. ABAP Objects is used in parallel with procedural techniques, and non-OO techniques — such as calling classical procedures or screens, or direct database access, for example — are coded inside ABAP Objects application classes. The hybrid programming model of SAP Web Application Server allows you to take advantage of enhanced SAP technology while preserving your investments in existing business processes, functionality, and data.

When Should You Use Which?

As this article will demonstrate, in most cases, ABAP Objects is the better choice, so once you've gotten the hang of using ABAP Objects, you'd be well advised to use the procedural technique only when absolutely necessary — with classical screen (Dynpro) programming, for example, which is not supported by ABAP Objects class pools (see the appendix for more on screen programming and ABAP Objects).

focus is on OO programming with ABAP Objects, and how it compares to the well-known procedural ABAP programming style. We will use the implementation of a simple bank account to illustrate the differences between programming with the two models.

Reason #1: Data Encapsulation

Encapsulating data (and functionality) within components makes it much easier to modify programs. Instead of putting all the data and functions of an entire application into one large program, you simply include everything a component of the application needs to function (i.e., the data and the corresponding procedures) in an object (e.g., a function pool in procedural programming or a class pool in ABAP Objects), and expose only the procedures that manipulate that data through a stable interface. If an encapsulated component has a well-defined interface, and can only be accessed through this interface, this conceals its internal structure and allows you to make changes without affecting the other components that make up the application (more on this in "Reason #4: Interfaces"). Let's take a look at how encapsulation works in procedural ABAP, and how it works in ABAP Objects.

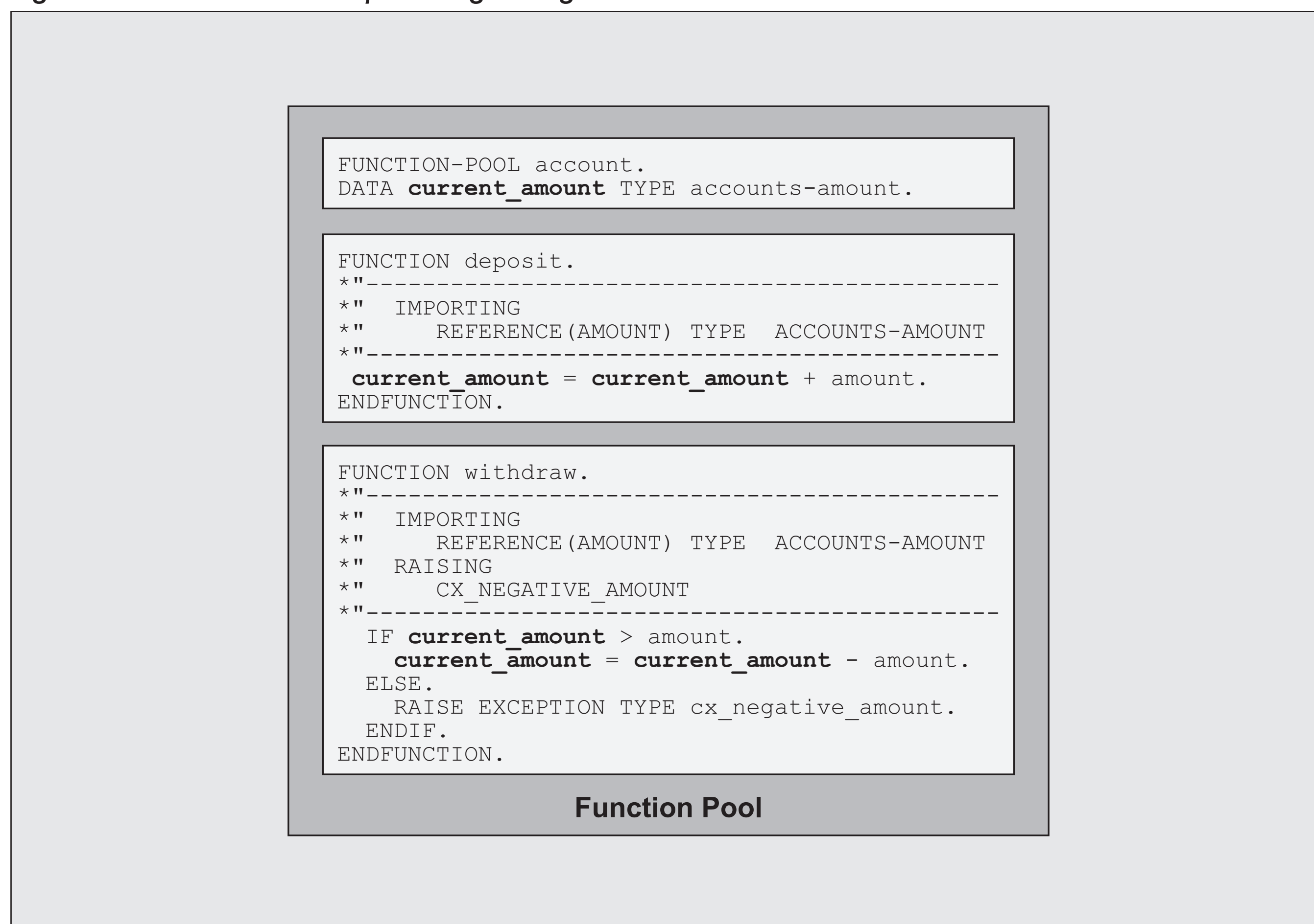
The Procedural Approach to Data Encapsulation

There are two kinds of data involved in the procedural programming model:

- **Global data**, which is declared in a global declaration section of the program and can be addressed from anywhere in the rest of the program

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 1 ***Encapsulating a Single Bank Account in a Function Pool***



- **Local data**, which can be addressed only within the particular procedure (subroutine or function module) in which it is declared

The lifetime of global data is bound to the runtime of the program, while the lifetime of local data is bound to the runtime of its procedure. Due to its limited lifetime, local data is not suited for encapsulation; it serves merely as a helper for internal activities inside its procedure. As long as you are not using *TABLES* statements or the *COMMON PART* addition to the *DATA* statement, which declare interface work areas, the global data of an ABAP program is not visible outside of the program; only the processing blocks of the program can work with this data. Therefore, the only level of data encapsulation that is possible in procedural programming is the ABAP program itself. **Figure 1** shows an example of how you could encapsulate a simple bank account by implementing it in a function pool.

The function pool encapsulates the data (*current_amount*) of an account, and the function modules *deposit* and *withdraw* work with the data of that account. While this kind of encapsulation works well for one account, the limitations of the procedural approach quickly become apparent when you want to work with multiple accounts and model the interactions between them. Using this technique, the only way to maintain real data encapsulation is to create a function pool and function modules for each account you want to work with, each with its own unique

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

name, which is simply impracticable. The work-around is to encapsulate all of the accounts in a single function pool, as shown in **Figure 2**.

Now the *accounts* function pool encapsulates the data of multiple accounts in an internal table (*account_tab*), where each account is identified via a unique business key ID. The internal table is filled when the function pool is loaded into memory (instantiated) and its content is shared by the function modules of the *accounts* function pool. The function modules *deposit* and *withdraw* work with one line of internal table *account_tab*, which is identified via the table key ID, while a new function module *transfer* uses two lines of *account_tab* to allow the interaction between two accounts (a transfer by processing a withdrawal and a deposit). Note that we have paid a price for the ability to work with multiple accounts, however — we have increased the complexity of the function module's interface by adding parameters to identify each account via its unique business key ID.

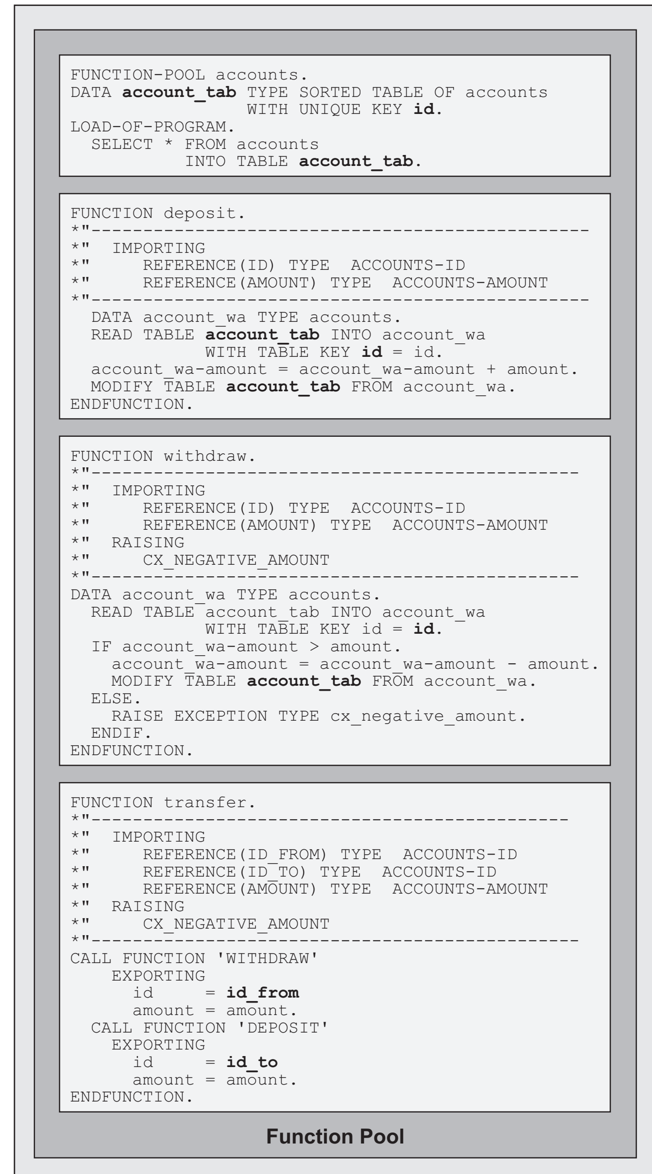
The Object-Oriented Approach to Data Encapsulation

The OO programming model also uses two kinds of data:

- The **attributes** of a class instance (an object)
- The **local data** in the class's methods

The lifetime of instance attributes[3] is bound to the lifetime of the object, while the lifetime of local data is bound to the runtime of the method. Local data, therefore, plays the same role in OO programming that it does in procedural programming — it serves as a helper for its method and is not appropriate for encapsulation. The suitable data encapsulation level

Figure 2 ***Encapsulating Multiple Bank Accounts in a Function Pool***



```
FUNCTION-POOL accounts.
DATA account_tab TYPE SORTED TABLE OF accounts
                 WITH UNIQUE KEY id.
LOAD-OF-PROGRAM.
  SELECT * FROM accounts
           INTO TABLE account_tab.
```

```
FUNCTION deposit.
*"-----------------------------------------------
*"  IMPORTING
*"     REFERENCE(ID) TYPE  ACCOUNTS-ID
*"     REFERENCE(AMOUNT) TYPE  ACCOUNTS-AMOUNT
*"-----------------------------------------------
  DATA account_wa TYPE accounts.
  READ TABLE account_tab INTO account_wa
             WITH TABLE KEY id = id.
  account_wa-amount = account_wa-amount + amount.
  MODIFY TABLE account_tab FROM account_wa.
ENDFUNCTION.
```

```
FUNCTION withdraw.
*"-----------------------------------------------
*"  IMPORTING
*"     REFERENCE(ID) TYPE  ACCOUNTS-ID
*"     REFERENCE(AMOUNT) TYPE  ACCOUNTS-AMOUNT
*"  RAISING
*"     CX_NEGATIVE_AMOUNT
*"-----------------------------------------------
DATA account_wa TYPE accounts.
  READ TABLE account_tab INTO account_wa
             WITH TABLE KEY id = id.
  IF account_wa-amount > amount.
    account_wa-amount = account_wa-amount - amount.
    MODIFY TABLE account_tab FROM account_wa.
  ELSE.
    RAISE EXCEPTION TYPE cx_negative_amount.
  ENDIF.
ENDFUNCTION.
```

```
FUNCTION transfer.
*"-----------------------------------------------
*"  IMPORTING
*"     REFERENCE(ID_FROM) TYPE  ACCOUNTS-ID
*"     REFERENCE(ID_TO) TYPE  ACCOUNTS-ID
*"     REFERENCE(AMOUNT) TYPE  ACCOUNTS-AMOUNT
*"  RAISING
*"     CX_NEGATIVE_AMOUNT
*"-----------------------------------------------
CALL FUNCTION 'WITHDRAW'
    EXPORTING
      id     = id_from
      amount = amount.
  CALL FUNCTION 'DEPOSIT'
    EXPORTING
      id     = id_to
      amount = amount.
ENDFUNCTION.
```

Function Pool

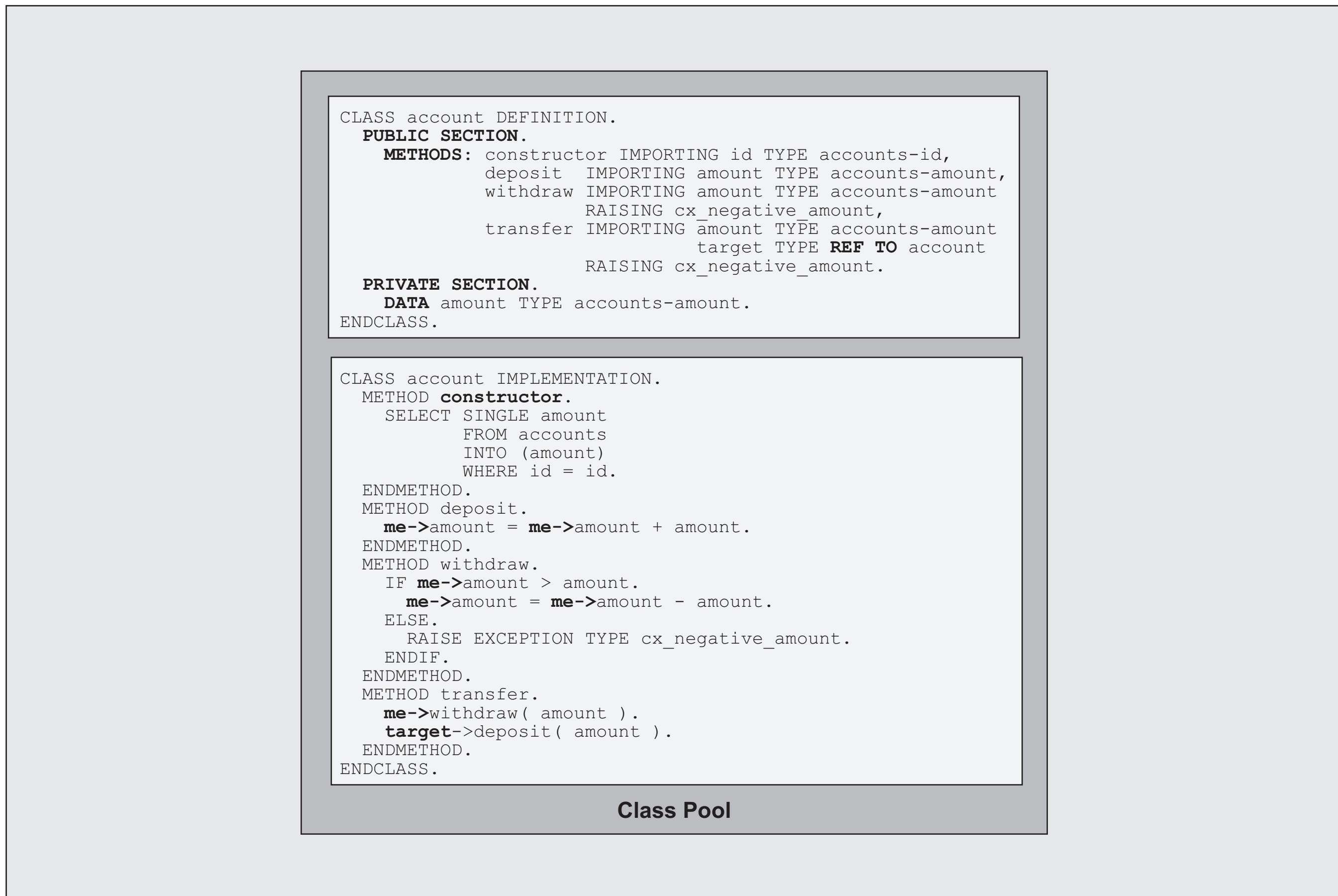
with ABAP Objects is the class.[4] Each attribute of a class is explicitly specified as either visible only inside the class or its subclasses ("private" or "protected," respectively), or visible from outside the class

[3] You can also declare static attributes inside a class. Static attributes follow the same visibility rules as instance attributes, and are bound to the class itself rather than an instance of the class (an object). Instance attributes play a far more important role in OO programming, so we will focus on them exclusively in this article.

[4] Only a class can be encapsulated in ABAP Objects, never an instance of a class (an object). You can access the private components of an object from another object if both are instantiated from the same class.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 3 **Encapsulating a Bank Account in a Class Pool**



```
CLASS account DEFINITION.
  PUBLIC SECTION.
    METHODS: constructor IMPORTING id TYPE accounts-id,
             deposit  IMPORTING amount TYPE accounts-amount,
             withdraw IMPORTING amount TYPE accounts-amount
                      RAISING cx_negative_amount,
             transfer IMPORTING amount TYPE accounts-amount
                                target TYPE REF TO account
                      RAISING cx_negative_amount.
  PRIVATE SECTION.
    DATA amount TYPE accounts-amount.
ENDCLASS.
```

```
CLASS account IMPLEMENTATION.
  METHOD constructor.
    SELECT SINGLE amount
           FROM accounts
           INTO (amount)
           WHERE id = id.
  ENDMETHOD.
  METHOD deposit.
    me->amount = me->amount + amount.
  ENDMETHOD.
  METHOD withdraw.
    IF me->amount > amount.
      me->amount = me->amount - amount.
    ELSE.
      RAISE EXCEPTION TYPE cx_negative_amount.
    ENDIF.
  ENDMETHOD.
  METHOD transfer.
    me->withdraw( amount ).
    target->deposit( amount ).
  ENDMETHOD.
ENDCLASS.
```


("public") — i.e., the attribute is either part of the "inside" of the class or part of the interface to the "outside" of the class. **Figure 3** shows how you can encapsulate the bank account example by implementing it in a class pool.[5]

In the definition part of the *account* class, its attributes and methods are defined in the appropriate visibility sections (*PUBLIC* and *PRIVATE*). The definition of a class explicitly unites functionality with its data. In principle, the function pool of Figure 1 implicitly does the same, but as you saw, this approach has its limitations when it comes to interaction between function modules, and for this reason function pools are generally used as containers for function modules that do not share data. In an ABAP Objects class, the *PUBLIC* section clearly defines the interface of the class to the outside world.[6] In the example in Figure 3, the data that defines the state of the class's objects is held private, while the public interface consists of the methods that manipulate that data.

The class in Figure 3 unites the simplicity of the approach shown in Figure 1 with the functionality of the approach shown in Figure 2. The implementation of the *deposit* and *withdraw* methods is the same as for the function modules in Figure 1. Since you can

[5] A class pool is an ABAP program that contains a global class. It serves the same purpose for OO programming that a function pool serves for procedural programming.

[6] In a function pool, the function modules are implicitly public, while all other components (procedures and data) are private. You cannot hide function modules, and you cannot publish other components.

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

instantiate many objects of a class (see "Reason #2: Instantiation"), the class only has to encapsulate the data for one account — the constructor allows you to initialize each account individually with its own variation of the data. Since each object works on its own data (shown in Figure 3 by explicitly using the optional self-reference *me*), there is no need for additional parameters (only the constructor needs a business key, and just once, to retrieve the data from a database table). And last, but not least, since one object can call the methods of other objects, the problem of interacting accounts is solved in a most elegant way using the *transfer* method, as shown in Figure 3, which is much simpler than the function module in Figure 2.

Figure 4 Implicit Instantiation of a Function Pool

```
DATA: id1(8) TYPE n,
      id2(8) TYPE n,
      amnt   TYPE p DECIMALS 2,
      exc_ref TYPE REF TO cx_negative_amount,
      text TYPE string.
TRY.
    id1 = ...
    id2 = ...
    amnt = ...
    CALL FUNCTION 'TRANSFER'
      EXPORTING
        id_from = id1
        id_to   = id2
        amount  = amnt.
    CATCH cx_negative_amount INTO exc_ref.
    text = exc_ref->get_text( ).
    MESSAGE text TYPE 'I'.
ENDTRY.
```

ABAP Program

Reason #2: Instantiation

Object instantiation is a key feature of OO programming. With ABAP Objects, you can explicitly instantiate multiple variations of a single class, which enables programmers to directly address an object and tell it to do something (address an account and deposit an amount there, or address a window and tell it to open or close, for example).

In procedural ABAP programming, a program is implicitly instantiated when it is loaded into memory. Such an instance cannot be handled by programmers explicitly, which necessitates the separation of data and functions — instead of working directly with software objects, the programmer tells a function which data it should alter (to which account a deposit should be made or which window should be opened or closed, for example). In the next sections, we'll look at the differences between these two types of instantiation.

The Procedural Approach to Instantiation

Many developers might not be aware that something similar to object instantiation takes place in procedural programming. Programs themselves are instantiated implicitly when they are loaded into memory, because the program itself is called (via *SUBMIT*, *CALL TRANSACTION*, etc.) or one of the program's procedures is called from another program (via *CALL FUNCTION*).

Figure 4 shows a call of function module *transfer* in the function pool *accounts* from Figure 2. The two accounts are identified via their business keys.

If the statement *CALL FUNCTION* is the first call of a function module in function pool *accounts*, the function pool is loaded into (instantiated in) the internal session of the calling program and the event *LOAD-OF-PROGRAM* is raised. The respective event block serves as a constructor for the function pool instance. If the same or another ABAP program in the same internal session had previously called a function module from the *accounts* function pool, the function pool would already be loaded into (instantiated in) the internal session. Since a program or procedure is loaded into memory only once, the function module would work with the global data of the already-existing function pool instantiation.

The disadvantages of this kind of instantiation for data encapsulation (refer back to the section "Reason #1: Data Encapsulation") are:

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

- You cannot control the moment of instantiation (function modules in a function pool can be called from anywhere, at any time, in a large application). Therefore, when a program or procedure calls a function module, there is no way to know if the global data of the function pool might have been changed by another program or procedure.

- You cannot delete an instance from memory, because the lifetime of an instance is bound to the runtime of the main program in memory. If you want to work only temporarily with a large amount of global data in a function pool, you must clear the data yourself after finishing your work. Furthermore, as existing function pools tend to be containers for many function modules, you strain the program memory until the end of its runtime, even if you use only a single, small function module from the pool for a short period of time.

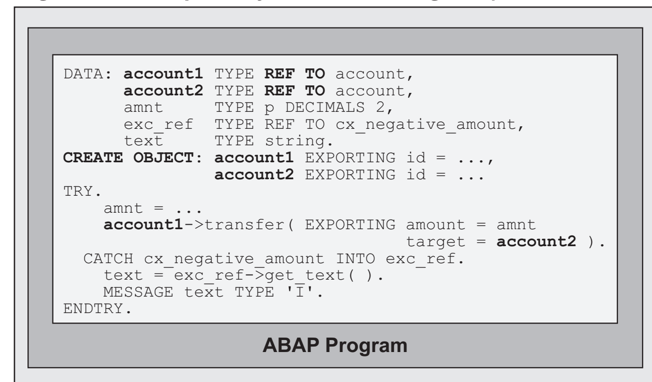
- You can create only one instance of each program in memory. As we have seen already in Reason #1, this prevents you from fully exploiting the data encapsulation capabilities of function pools.

Please note that when a function pool does not handle its global data properly, these disadvantages can make the use of global data in function modules rather dangerous. The state of the global data depends on the sequence of function calls, which might easily become undefined in long-running transactions, where the function modules of a function pool can be called at any time from different parts of the program.[7]

The Object-Oriented Approach to Instantiation

It's already clear from the name "object-oriented"

Figure 5 Explicitly Instantiating Objects



```
DATA: account1 TYPE REF TO account,
      account2 TYPE REF TO account,
      amnt     TYPE p DECIMALS 2,
      exc_ref  TYPE REF TO cx_negative_amount,
      text     TYPE string.
CREATE OBJECT: account1 EXPORTING id = ...,
               account2 EXPORTING id = ...
TRY.
    amnt = ...
    account1->transfer( EXPORTING amount = amnt
                                  target = account2 ).
  CATCH cx_negative_amount INTO exc_ref.
    text = exc_ref->get_text( ).
    MESSAGE text TYPE 'I'.
ENDTRY.
```


that the existence of objects is key to OO programming. Objects are instances of classes, and are created via the statement *CREATE OBJECT*. There is no implicit instantiation of objects in ABAP Objects — you work with an object explicitly using reference variables that point to the object. You might work with objects that you have not created yourself, in which case a reference to that object is explicitly passed to you from somewhere else, like a factory method, for example, or via a parameter interface. You can instantiate many objects from a single class, each with its own identity and contents in its instance attributes.

The lifetime of an object is controlled by its users (other objects, programs, procedures, etc.), and as long as reference variables are pointing to an object, it stays in memory. If no reference variable is pointing to an object, it is deleted automatically by the ABAP Objects garbage collector. Like instances of programs, objects live in the internal session of the main ABAP program. As of Release 6.40, you can also create "Shared Objects" in the shared memory of an application server, which can be used by all programs of each server.[8]

Figure 5 shows the instantiation of two *account* objects from the class *account* in Figure 3.

---

[7] A function pool is loaded into memory only once. All programs loaded into the internal session of a main program work with the same instance of a function pool when they call function modules of that pool. Therefore, as a caller of an external procedure, you can never be sure if the global data of a function pool you have previously loaded hasn't been changed by a procedure in the call stack that called a function module in that pool.

[8] An upcoming *SAP Professional Journal* article will cover Shared Objects in detail.

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Two reference variables, *account1* and *account2*, are defined as explicit handles for objects of class *account*. In the statement *CREATE OBJECT*, a business key is passed to the constructor of each object. After creating the object, the key is no longer necessary; the object itself represents the respective bank account. You can call the method *transfer* for one account and directly specify the target account via its reference.

A comparison with Figure 4 shows that the OO method is much more straightforward than working with function modules. While in Figure 4 you must tell the function the data it should work with, in Figure 5 you simply call a method of an object and it works automatically with its own data. Note also that we work with class-based exceptions in both figures, which is recommended as of Release 6.10.[9] While the use of reference variable *exc_ref* appears a bit alien in a procedural programming environment, because it introduces objects and object handling to a non-object world, it is a natural part of OO programming.

Reason #3: Code Reuse

Code reuse is an important instrument for lowering the cost of software development and maintenance. Instead of programming the same functions or variants of functions over and over again, you store and maintain commonly used code in central libraries that can be reused across your programs. We'll examine the different ways that ABAP Objects and procedural ABAP enable you to reuse code in your programs in the next sections.

The Procedural Approach to Code Reuse

In procedural programming, there is no dedicated mechanism for code reuse. You can define generalized procedures, but if you want to reuse general code for special purposes, you will need to create large function pools, define extensive parameter interfaces for the function modules, and subdivide tasks via logical conditions, with huge *CASE* switches, for example.

Let us consider the simple task of enhancing our example by creating specialized accounts — a checking account and a savings account — from our example account in Figure 2. **Figure 6** shows a possible modification of the account's *withdraw* function to accommodate reuse.

In order to preserve the function module *withdraw* in the function pool *account*, its parameter interface must be enhanced with an additional steering parameter, *kind*, that denotes the account type.

Furthermore, an additional exception must be defined, which is raised when an incorrect value is passed for the *kind* parameter.

Finally, to implement the different handling of withdrawals from checking or savings accounts, the function module is submodularized in two subroutines (*withdraw_from_checking_account* and *withdraw_from_savings_account*), which are called from a *CASE* list.

The Object-Oriented Approach to Code Reuse

In OO programming, code reuse is supported by the concept of "inheritance," which allows you to create specialized subclasses by deriving them from a generalized superclass. The benefits of inheritance are small classes compared to large function pools, minimal parameter interfaces of methods, and — most important — polymorphism, which enables a user to continue working with the superclass interface regardless of where it is implemented (in subclasses). All in all, inheritance enhances the modeling capabilities for your software tremendously.

[9] For more on this subject, see the article "A Programmer's Guide to the New Exception-Handling Concept in ABAP" in the September/October 2002 issue of *SAP Professional Journal*.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 6 **Reusing Code with Procedural Programming**

```
FUNCTION withdraw.
*"------------------------------------------------------------
*"  IMPORTING
*"     REFERENCE(ID) TYPE  ACCOUNTS-ID
*"     REFERENCE(KIND) TYPE  C DEFAULT 'C',
*"     REFERENCE(AMOUNT) TYPE  ACCOUNTS-AMOUNT
*"  RAISING
*"      CX_NEGATIVE_AMOUNT
*"      CX_UNKNOWN_ACCOUNT_TYPE
*"------------------------------------------------------------
  CASE kind.
    WHEN 'C'. " Checking account
      PERFORM withdraw_from_checking_account USING id amount.
    WHEN 'S'. " Savings account
      PERFORM withdraw_from_savings_account USING id amount.
    WHEN OTHERS.
      RAISE EXCEPTION TYPE cx_unknown_account_type.
  ENDCASE.
ENDFUNCTION.
```

```
FORM withdraw_from_checking_account
     USING l_id TYPE accounts-id
           l_amount TYPE accounts-amount.
  DATA account_wa TYPE accounts.
  READ TABLE account_tab INTO account_wa WITH TABLE KEY id = l_id.
  account_wa-amount = account_wa-amount - l_amount.
  MODIFY TABLE account_tab FROM account_wa.
  IF account_wa-amount < 0.
    ... " Handle debit balance
  ENDIF.
ENDFORM.
```

```
FORM withdraw_from_savings_account
     USING l_id TYPE accounts-id
           l_amount TYPE accounts-amount
     RAISING cx_negative_amount.
  DATA account_wa TYPE accounts.
  READ TABLE account_tab INTO account_wa WITH TABLE KEY id = l_id.
  IF account_wa-amount > l_amount.
    account_wa-amount = account_wa-amount - l_amount.
    MODIFY TABLE account_tab FROM account_wa.
  ELSE.
    RAISE EXCEPTION TYPE cx_negative_amount.
  ENDIF.
 ENDFORM.
```

Function Pool

Figure 7 is a modified version of the *account* class in Figure 3. It has been generalized in order to serve as a superclass for two specialized subclasses that will be derived from it — a checking account and a savings account.

As you can see, the visibility of attribute *amount* has changed from *PRIVATE* to *PROTECTED* to make it available to subclasses, and the implementation of method *withdraw* has been generalized; it no longer throws a specific exception. Both changes have no effect on the outside user (i.e., a program or other objects), which is one of the prerequisites for polymorphism.

Figure 7 **A Generalized Account Superclass**

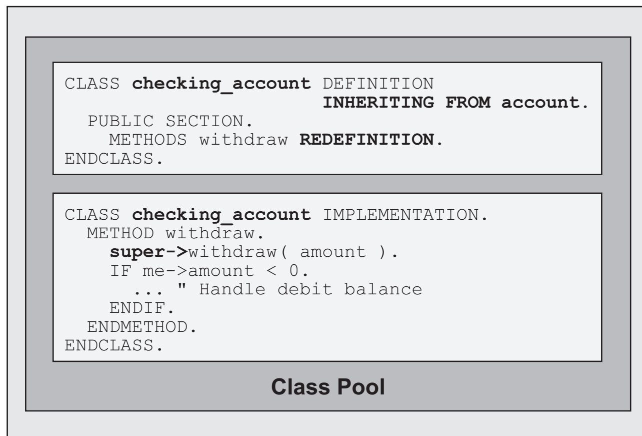
```
CLASS account DEFINITION.
  PUBLIC SECTION.
    ...
  PROTECTED SECTION.
    DATA amount TYPE accounts-amount.
ENDCLASS.
```

```
CLASS account IMPLEMENTATION.
  ...
  METHOD withdraw.
    me->amount = me->amount - amount.
  ENDMETHOD.
  ...
ENDCLASS.
```

Class Pool

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

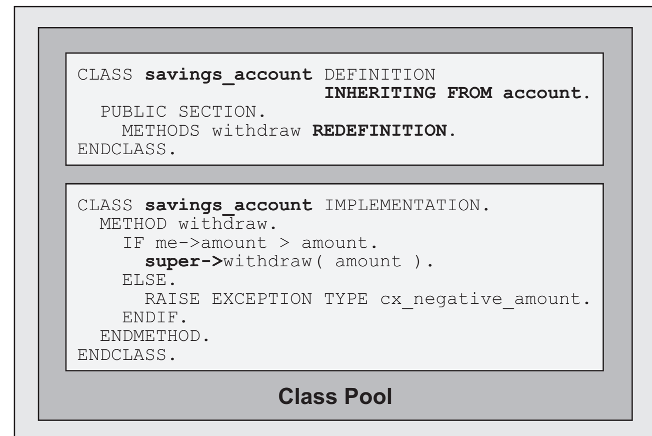
Figure 8 A Checking Account Subclass Derived from the Account Superclass



```
CLASS checking_account DEFINITION
                       INHERITING FROM account.
  PUBLIC SECTION.
    METHODS withdraw REDEFINITION.
ENDCLASS.
```

```
CLASS checking_account IMPLEMENTATION.
  METHOD withdraw.
    super->withdraw( amount ).
    IF me->amount < 0.
      ... " Handle debit balance
    ENDIF.
  ENDMETHOD.
ENDCLASS.
```


Figure 9 A Savings Account Subclass Derived from the Account Superclass



```
CLASS savings_account DEFINITION
                      INHERITING FROM account.
  PUBLIC SECTION.
    METHODS withdraw REDEFINITION.
ENDCLASS.
```

```
CLASS savings_account IMPLEMENTATION.
  METHOD withdraw.
    IF me->amount > amount.
      super->withdraw( amount ).
    ELSE.
      RAISE EXCEPTION TYPE cx_negative_amount.
    ENDIF.
  ENDMETHOD.
ENDCLASS.
```

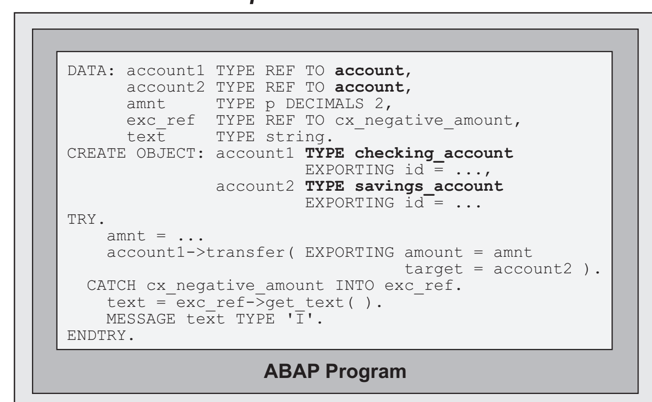

Figure 8 and **Figure 9** show the specialized subclasses *checking_account* and *savings_account* derived from the generalized superclass *account*.

By using *INHERITING FROM* in the class definition, both subclasses contain the same components and the same public interface as the *account* superclass. By using *REDEFINITION* for the method *withdraw*, the subclasses can provide a more specialized (appropriate) implementation of the method than the superclass. In both method redefinitions, the general implementation of the superclass is reused by calling the method with the prefix *super->* and both methods are enhanced with special features for the respective account type — handling the debit balance for the checking account and raising an exception for a negative amount in the savings account, for example.

Figure 10 demonstrates how the specialized subclasses can be used instead of the *account* objects shown in Figure 5.

Compare Figure 10 to Figure 5. The only difference between the two is the *TYPE* additions in the *CREATE OBJECT* statement, so that one object is created from subclass *checking_account* and the other from subclass *savings_account*. Note that the reference variables of both objects, *account1* and *account2*, are still typed with the *account* superclass. If Figure 10 instead referenced objects created elsewhere, even in other programs, for example, you would not see any difference at all. From the point of view of the user (a program, a procedure, etc.), the coding in Figure 5 and Figure 10 work the same — the system, however, automatically executes the right implementation of method *withdraw* (when called from method *transfer*) for each account type. That is what polymorphism is all about! The benefits are obvious when you look at the parameter interface of the function module *withdraw* in Figure 6. To allow transfers between different types of accounts

Figure 10 Instantiating Objects Using the Specialized Subclasses



```
DATA: account1 TYPE REF TO account,
      account2 TYPE REF TO account,
      amnt     TYPE p DECIMALS 2,
      exc_ref  TYPE REF TO cx_negative_amount,
      text     TYPE string.
CREATE OBJECT: account1 TYPE checking_account
                        EXPORTING id = ...,
               account2 TYPE savings_account
                        EXPORTING id = ...
TRY.
    amnt = ...
    account1->transfer( EXPORTING amount = amnt
                                  target = account2 ).
  CATCH cx_negative_amount INTO exc_ref.
    text = exc_ref->get_text( ).
    MESSAGE text TYPE 'I'.
ENDTRY.
```


©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

with the procedural approach, your implementation of the *transfer* function module would have to consider all possible transfers to and from accounts, which in our overly simplified example is already 2 × 2. And each time you introduce a new account type, our maintenance efforts increase exponentially. With ABAP Objects, on the other hand, we didn't touch the *transfer* method at all in our simple example.

Reason #4: Interfaces

Generally speaking, interfaces provide you with access from one programming context to the data and functions of another programming context. In a world of modularization and interaction, interfaces define a point of contact between components. ABAP Objects provides standalone interfaces between a component and its caller, which enables programmers to decouple classes from their users (programs, objects, etc.). Instead of standalone interfaces, procedural ABAP provides implicit functional interfaces for programs and explicit data interfaces for programs and procedures. Let's take a closer look at the differing uses of interfaces in the two programming models.

The Procedural Approach to Interfaces

Procedural programming offers limited support for interfaces:

- Global data declared with the *TABLES* statement or the *COMMON PART* addition to the *DATA* statement are "data interfaces" between programs. In classical ABAP screen programming, interface work areas declared with *TABLES* constitute the (implicit) interface between screens (Dynpros) and programs. Logical databases also use this type of implicit interface.

- Procedures that can be called from other programs are "functional interfaces" between programs. Function modules and subroutines are implicitly always public. By calling an external procedure, the calling program has access to the data of the procedure's main program.

Apart from the parameters and selection criteria of selection screens, which can be used as parameter interfaces when calling executable programs with the *SUBMIT* statement, there is no support for explicit interfaces between programs. When a developer changes the implicit interfaces of a program, there is always the danger of invalidating a user of the program that the developer is unaware of. And it is always a risky proposition to allow a caller unfettered access to the data of the main program.

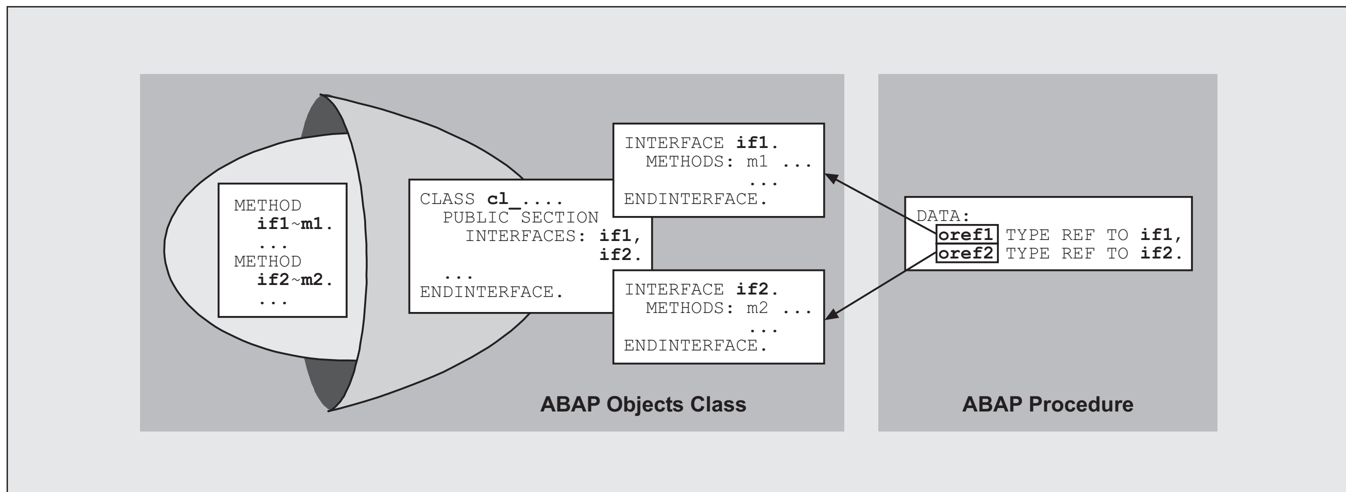
The Object-Oriented Approach to Interfaces

In OO programming, interfaces are defined explicitly by declaring class components in one of three sections (*PUBLIC*, *PROTECTED*, and *PRIVATE*) that restrict the visibility of the components defined there. Components declared in the *PUBLIC* section — i.e., data interfaces and functional interfaces — are visible to the outside world. Components declared in the other sections are either not accessible from the outside (*PRIVATE*), or accessible only if there is a certain relation — via inheritance (*PROTECTED*) or as a "friend" (*PROTECTED* or *PRIVATE*) — to the exposing class. Through inheritance, a subclass inherits the interface of its superclass, which can then be extended with the subclass's own components.

In order to support its stability, and therefore its (re)usability, the interface defined in the *PUBLIC* section can be factored out into one or more "standalone" descriptions of the class's interface, or parts of the class's interface. Such standalone interfaces do not include any implementation; rather, they include some or all public properties of a class. Standalone interfaces can be used by more than one class and usually describe one special aspect of a class, such as unified handling for persisting attributes, for example. Furthermore, since they are independent of an actual implementation, independent characteristics of objects can be separated into

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 11 *Standalone Interfaces*



different interfaces. **Figure 11** illustrates the concept of standalone interfaces.

A standalone interface is defined between the statements *INTERFACE* and *ENDINTERFACE*.[10] It can contain exactly the same kinds of components as a class. Any class can include one or more interfaces, defined with the statement *INTERFACES* in its *PUBLIC* section. As a result, the components defined in the interface become part of the class's public interface. A class that includes a standalone interface must provide the service the interface promises to offer, so that whenever a user accesses the interface, the expected service is provided without the need for the user to know how it is implemented.

So how does this work? The basic principle is that you can declare reference variables of a certain interface type (see the data declaration at the right of Figure 11) that can point to objects of all classes that implement this interface. A user (program or object) that works with an interface reference variable pointing to an object need not care about the actual class of the object at all. It is enough for a user to know only about the properties of the interface. Because every class implementing this interface must implement all of the interface's components, the user can expect that the methods it is calling work as described. While each class implements an interface method according to its own needs — for example, the formula to calculate interest can differ among account types — the method name and parameters in the interface remain stable. Therefore, standalone interfaces are another foundation for polymorphism in OO programming, along with inheritance.[11] Accessing an object via an interface reference variable is conceptually the same as accessing a subclass object via a superclass reference variable.

Figure 12 and **Figure 13** show two classes that normally have nothing in common: our now-familiar example class *account*, and a new class, *customer*, which could be part of the same or another application component; it makes no difference. Both classes include the interface *if_serializable_object*, which is available in the class library as of Release 6.10. This interface allows you to serialize the attributes of an object into an XML document. By including the *if_serializable_object* interface, both classes inherit the serializable property and offer this property to the outside user.

Figure 14 shows a program working with objects of both classes: *account* and *customer*. In this program, an internal table, *serializable_objects,* is defined

[10] A standalone interface can also be defined globally in the Class Builder.

[11] In fact, the concept of standalone interfaces can be seen as a kind of multiple inheritance from totally abstract classes (abstract classes without even a partial implementation of methods).

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 12 The Account Class Inherits the Serializable Property from the Interface

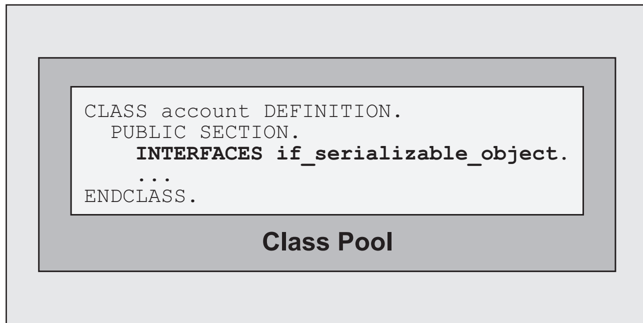
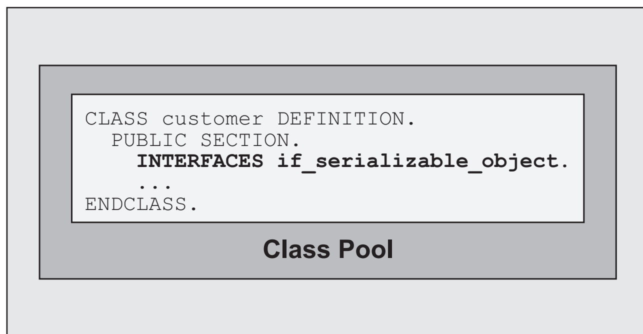


Figure 13 The Customer Class Inherits the Serializable Property from the Interface

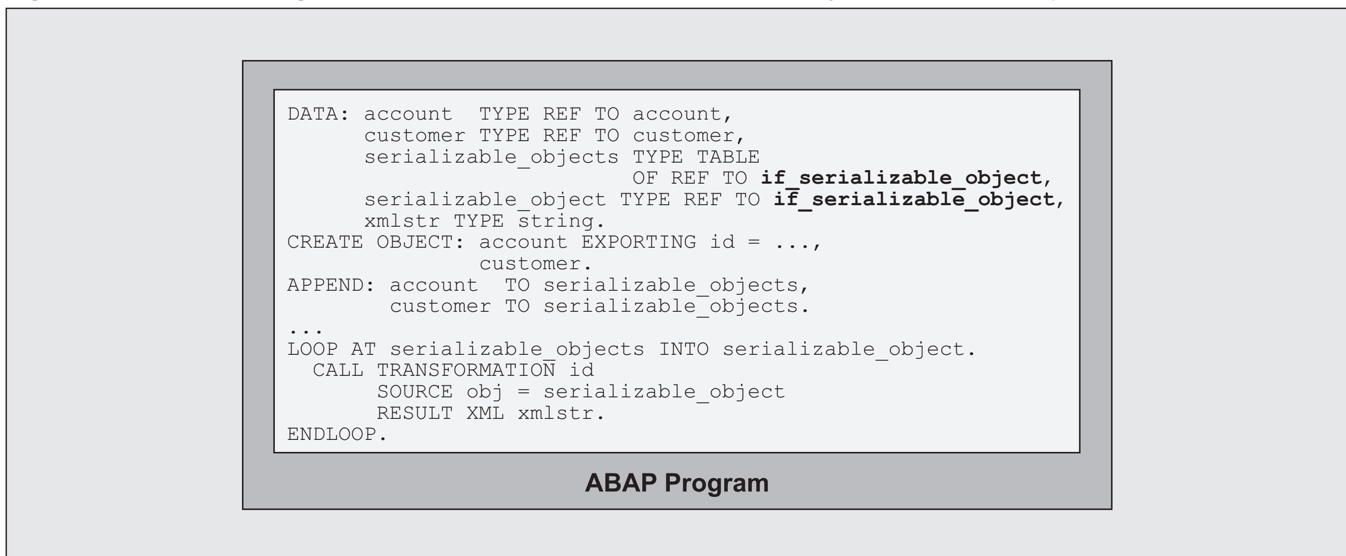


as a collection of interface reference variables of type *if_serializable_object*. Assume that at some point in the program, the internal table is processed within a *LOOP*, where all objects currently referenced by the internal table are serialized via the *CALL TRANSFORMATION* statement. The actual classes of the objects are of no concern inside of the loop; only the property that defines the objects as serializable is relevant. Standalone interfaces in combination with interface reference variables allow you to work with the select aspects of an object you need for a specific purpose without having to be concerned about other properties. In comparison, when working with function modules, you often have to pass parameters that are not relevant to the task at hand.

Reason #5: Events

When an event occurs — for example, the event "mouse click occurred" in the user interface or "new account created" in a business application — it needs to be made known to the interested event handlers. Events and event handling can be compared to procedure calls: raising an event corresponds to calling a procedure, and handling an event corresponds to the called procedure (the "callee"). The main difference

Figure 14 Using a Standalone Interface to Process Only Serializable Objects



©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

between procedure calls and the "event" concept is that a procedural caller and callee are tightly coupled; with events they are decoupled, which means that any potential handler can handle a raised event but is not required to do so. While procedural ABAP programs can handle events from the runtime environment, ABAP Objects provides a complete events concept that allows classes and objects to both raise and handle events. The next sections compare how procedural ABAP and ABAP Objects support events and event handling.

The Procedural Approach to Events

If you have performed procedural ABAP development, then you are likely well aware of the event concept. You can't execute an ABAP program without handling events. In order to be executed by *SUBMIT*, a program must contain at least a handler for the reporting event *START-OF-SELECTION*, which is triggered by the runtime environment. The screen flow logic must contain handlers for the PBO and PAI events triggered by the runtime environment during screen processing. Furthermore, you can handle *AT SELECTION-SCREEN* events from selection screens and *AT LINE-SELECTION* and *AT USER-COMMAND* events from ABAP lists.

One of the problems with procedural ABAP events is their implicitness. With only two exceptions (*PUT*, which triggers the *GET* event from logical database programs, and *SET USER-COMMAND*, which triggers the *AT USER-COMMAND* event), procedural ABAP events cannot be triggered from an ABAP program. You have to know the program's particular processes (the reporting process, the dialog process, the selection screen process, the list process, etc.) in the ABAP runtime environment in order to understand your program flow so that you can implement each handler in an appropriate way. Furthermore, there is a one-to-one relationship between handler and event. You cannot implement different handlers for one event, and you cannot deactivate a handler once it is implemented. Last, but not least, you cannot define and trigger your own events, meaning that procedural ABAP supports only a strong coupling between callers and procedures, which means that the moment you call a procedure, it must exist or a runtime error occurs.

The Object-Oriented Approach to Events

In ABAP Objects, events are explicitly declared as components of classes. When an event is declared in a class, it can be triggered explicitly by the statement *RAISE EVENT* within the methods of that class. In addition to notifying the runtime environment that an event occurred, the statement *RAISE EVENT* can pass additional parameters to the handlers that are interested in reacting to the event. Such a handler is just a method of some other (or the same) class, and its ability to react to an event is determined by a static declaration that it is an event handler for that specific event. During the runtime of a program, event handlers can be activated or deactivated dynamically at any time, and there can be multiple handlers for the same event at the same time.

The benefit of the event concept in ABAP Objects is the decoupling of the caller and the callee. Instead of calling a method directly, you can simply raise an event, and all handlers that are able to handle it (i.e., that are statically declared as an event handler for that event and are dynamically activated at runtime) handle it.[12] This is in contrast to normal procedure calls, where the caller has to know exactly what to call. The two stages of "publish and subscribe" — static declaration of event handlers and dynamic activation of the handling at runtime — give you flexibility when programming with events. **Figure 15** and **Figure 16** show how you can enhance our example class *account* with events.

In Figure 15, the subclass *checking_account* from Figure 8 is extended with event *advising_required*. This event is raised in the redefined method *deposit* when the amount exceeds a given limit. In Figure 16, another subclass, *advisor*, is defined with event handler method *receive_notification* for the

[12] Event handlers are executed sequentially. The order of execution depends on the order of activation.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 15 *Adding an Event*

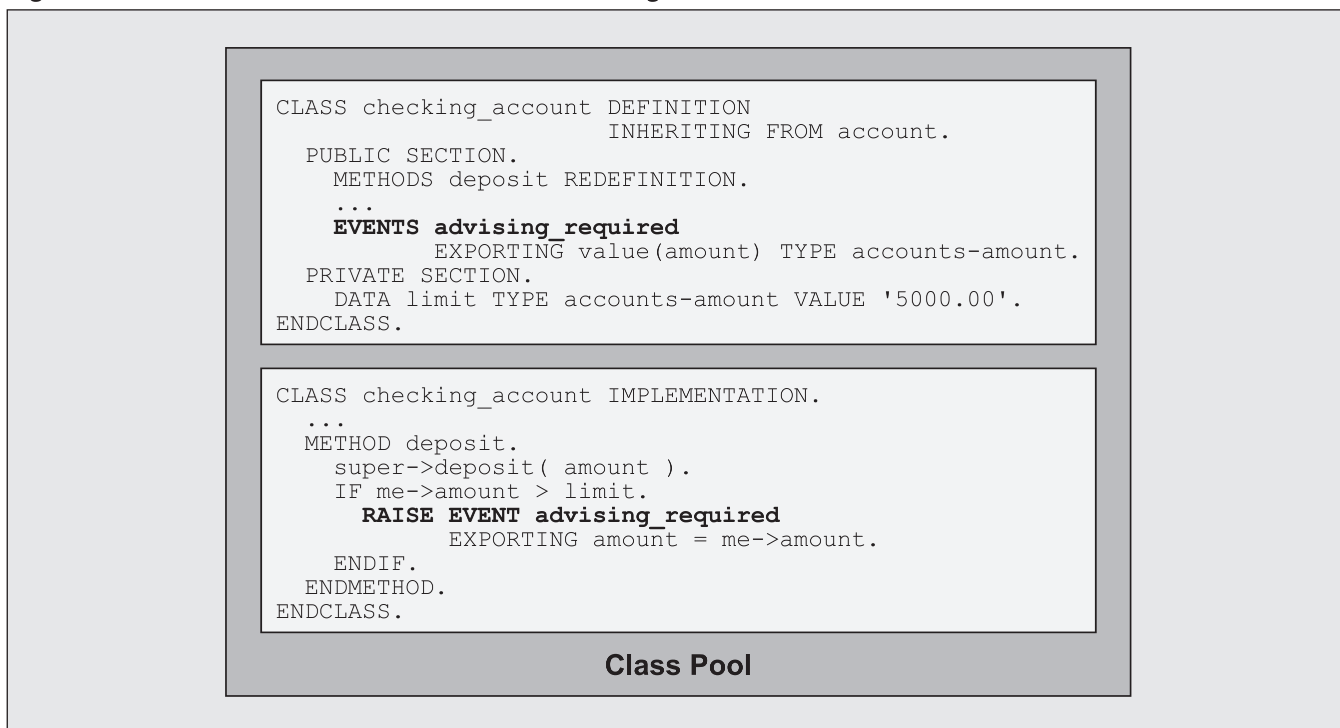
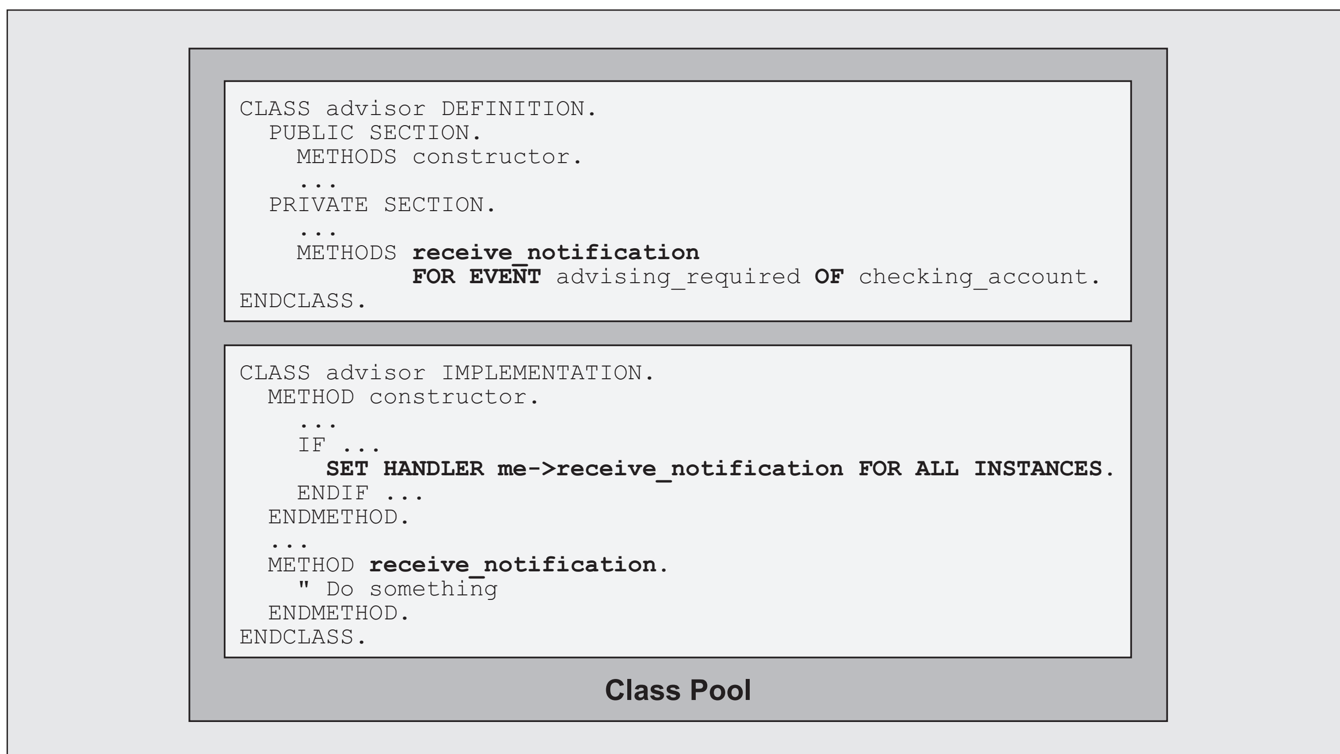


Figure 16 *Adding an Event Handler*



©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

advising_required event. During construction, an object of this class decides in an *IF* block whether to handle the event and for which accounts it will do so. In our example, an *advisor* object would react to events from all checking accounts. Of course, the syntax of the *SET HANDLER* statement also allows you to select particular accounts by specifying single objects after *FOR* instead of using *FOR ALL INSTANCES*. Therefore, it would be possible to subscribe to selected accounts only (e.g., to accounts of all customers that are assigned to a specific *advisor* object).

Are You a Believer Yet?

Since Release 4.6, ABAP has been a hybrid language that offers you the choice to stay in the familiar procedural programming model or switch to the enhanced ABAP Objects environment. Using ABAP Objects makes your daily development life much easier and more productive by adding the following features to your development arsenal:

- Classes that serve as templates for objects
- Objects (instances of classes) that are addressed by references
- Single inheritance
- Standalone interfaces
- Events that can be raised and handled by objects

In the preceding sections, you saw five specific ways that you can benefit from these additions. While these benefits are true for any OO language, ABAP Objects is a language optimized especially for business application programming. It is designed to be simpler than Java and C++, and to omit complex concepts that lack in benefits (such as multiple inheritance and destructors) or lead to errors (e.g., ABAP Objects uses a technique that circumvents the "diamond problem" of multiple inheritance[13]). It also implements powerful concepts such as events and event handling, which are available only via interfaces in other languages, as direct language elements.

[13] The diamond problem occurs when a class inherits instance variables or method implementations from two different subclasses of a superclass, and doesn't know which one to use.

Still hesitant to make the full leap to ABAP Objects? The hybrid nature of the ABAP runtime allows you to stick your toe in the water rather than dive all the way in, by using a combination of procedural ABAP and ABAP Objects. This capability not only protects your investment in your existing procedural development work, but it also offers you an opportunity to take advantage of ABAP Objects without having to fully adopt the OO programming model until you're ready. In the next sections, we'll show you three special reasons to use ABAP Objects to improve your ABAP programs.

Three Special Reasons ABAP Objects Is the Better ABAP

Up to now we have concentrated mainly on the benefits of using OO programming features and methodology. However, there are additional benefits to the ABAP programming language itself. In the next few sections, we'll take a look at how you can take advantage of ABAP Objects to improve your ABAP programs, and save yourself some valuable resource costs, even if you don't adopt a full OO approach to development.

Reason #1: ABAP Objects Is More Explicit, and Simpler to Use

Programming with ABAP Objects is simpler than procedural ABAP programming because it is more explicit. You have a much better feel for what lies below the surface. Take a look at the report example shown in **Figure 17**. Seems simple enough, right?

The truth is, you don't *really* know what's happening behind the scenes when this program is executed. The coding looks simple, but what's executed in the background is actually quite complicated. Check the documentation, and you will see that starting a report with *SUBMIT*, where the report is connected to a

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 17 Creating a Simple ABAP Report

```
REPORT simple_report.
NODES spfli.
GET spfli.
  WRITE: / spfli-carrid, spfli-connid ...
```

ABAP Program

logical database, automatically triggers a process consisting of about 10 steps, at least 1 loop, and the implicit processing of 2 screens. In the early days of ABAP, this was intended to provide the application developer with an automated, turnkey programming pattern. Automation is a great thing when you have an intuitive feel for what's taking place; it's a worrisome mystery when you don't. So as long as you are programming within a familiar realm, all is well. Venture outside your domain of expertise, and things can become enigmatic, for example:

- Some important interfaces are realized by global data, like the interface between screens and an ABAP program, and the interface between a logical database and a reporting program.
- Procedural ABAP programs are implicitly controlled by the runtime system — they are driven either from screens or from the reporting process after *SUBMIT*.

ABAP Objects, in comparison, is much simpler and less error-prone, because it is based on a few basic and orthogonal concepts[14]:

- Classes contain attributes and methods.
- Objects are instances of classes.
- Objects are addressed via references.
- Objects have clearly defined interfaces.

After learning the small set of statements you need for working with ABAP Objects, you can develop your programs such that you tell the system what it should do directly. The system's behavior is explicitly set by you, not implicitly determined by the runtime system. Instead of searching for and reading lengthy documentation about implicit system behavior, you simply read the programs themselves in order to understand their logic.

Figure 18 shows an example of how you can use ABAP Objects to transform a procedural-based

[14] A restricted set of independent concepts that can be combined in many meaningful ways.

Figure 18 Simplifying a Logical Database with ABAP Objects

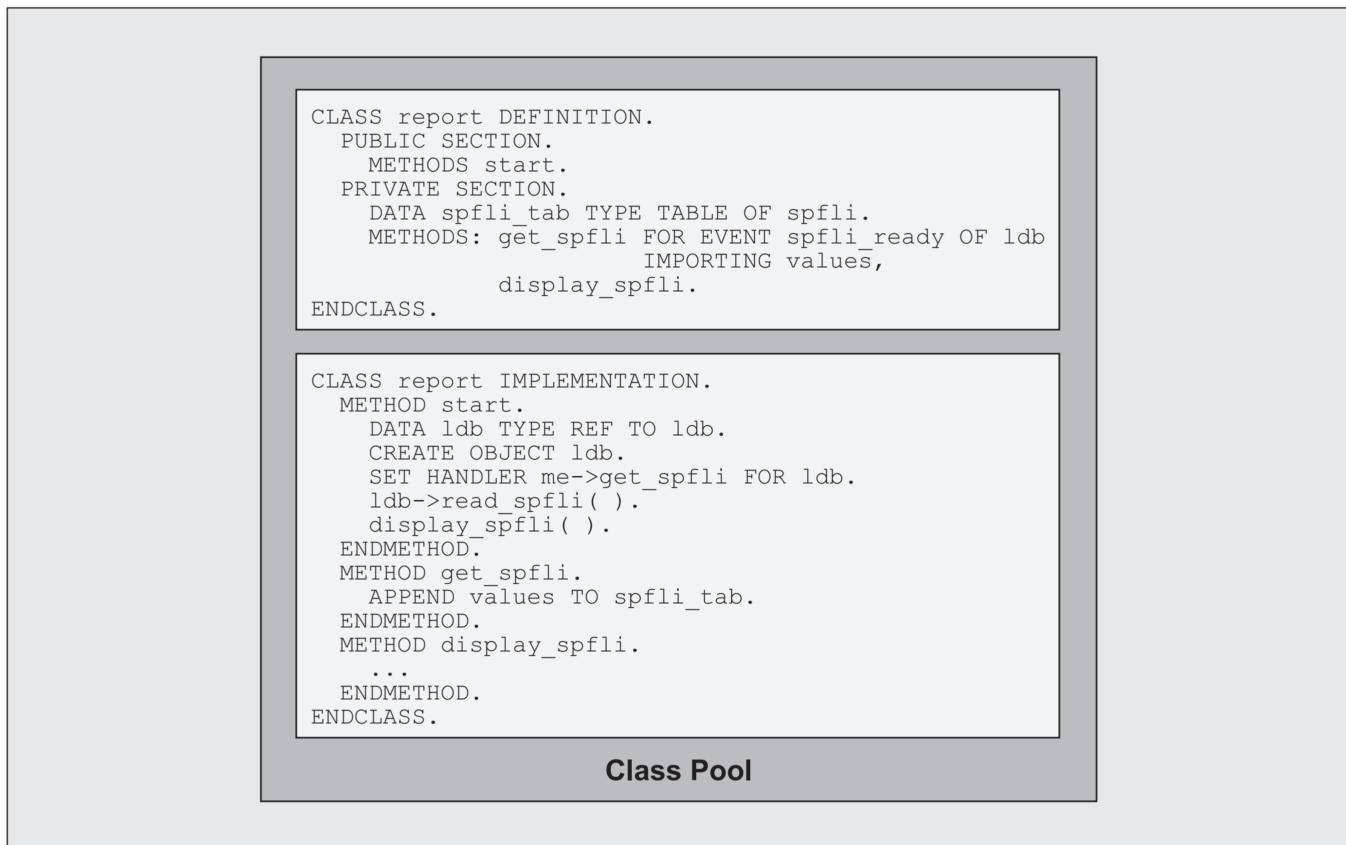
```
CLASS ldb DEFINITION.
  PUBLIC SECTION.
    METHODS read_spfli.
    EVENTS spfli_ready EXPORTING value(values) TYPE spfli.
  PRIVATE SECTION.
    DATA spfli_wa TYPE spfli.
ENDCLASS.
```

```
CLASS ldb IMPLEMENTATION.
  METHOD read_spfli.
    SELECT * FROM spfli
             INTO spfli_wa.
      RAISE EVENT spfli_ready EXPORTING values = spfli_wa.
    ENDSELECT.
  ENDMETHOD.
ENDCLASS.
```

Class Pool

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 19 Creating a Report from a Logical Database Simplified with ABAP Objects



implementation of a logical database to a class-based implementation that is much easier to understand.

The important thing to note about the class in Figure 18 — something that you don't find in procedural logical databases — is its explicit interface. You learn which services the class offers just by looking at the *PUBLIC* section. **Figure 19** shows a simple example of a class that creates a report using the class-based logical database from Figure 18.

In Figure 19, the method *start* replaces the event block *START-OF-SELECTION* that is implicit in Figure 17. In order to run the report, this method can, for example, be coupled to a transaction code and started as an OO transaction. Instead of the implicit event handling of Figure 17 via *GET*, the class in Figure 19 contains an explicit event handler method *get_spfli*. Furthermore, the data display is decoupled from the data handling in a method *display_spfli*.

If you compare Figure 19 with Figure 17, you see that everything you need to understand the concept is written in the coding. There is no "black magic" anymore. The comparison certainly dispels the misconception that simpler coding means less coding — that is not always the case. Keep in mind, however, that if you want to make the program in Figure 17 understandable to a non-ABAP reporting expert, you would need to add a lot of comment lines.

It should be clear from the examples here that programs implemented with ABAP Objects are easier to understand, easier to maintain, and more reliable, thanks to improved data encapsulation, parameter interfaces with secure typing,[15] and predictable behavior.

[15] The syntax check in ABAP Objects (more on this in the next section) forces you to specify a type (complete or generic) for each method or event parameter. The typing is checked at compile time, even when you call methods of global classes, and not postponed until runtime, as it is with function modules.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Reason #2: ABAP Objects Has a Stricter Syntax Check in Classes

Procedural ABAP has evolved over a long period of time, and as a result:

- Contains a large number of obsolete statements
- Supports overlapping concepts as well as highly specialized concepts
- Sometimes shows surprising implicit behavior
- Appears difficult to learn

Since no existing (i.e., pre-ABAP Objects) coding would be affected, with the release of ABAP Objects SAP took the opportunity to eliminate some of the peculiarities, obsolete elements, and surprise and mystery (or misery) of ABAP by placing restrictions on statements used inside of ABAP Objects classes, including:

- Prohibiting many obsolete statements and additions
- Requiring many implicit syntax completions to be explicit
- Detecting and preventing potentially incorrect data handling

> ✓ ***Note!***
>
> *Questionable constructs are still allowed outside of ABAP Objects classes to support the strict downward-compatibility requirement of ABAP, which relieves you from having to change your programs with every new ABAP release. On the other hand, it places a heavy burden on the language itself since it is full of obsolete constructs that have been replaced by better concepts, but cannot be removed, as they are widely used in existing programs. If you use any language elements from Figure 20 outside of a class, the syntax check can do no more than issue a warning not to use the obsolete language elements. To get rid of the warnings, we advise using the recommended replacements from the documentation outside of classes as well as inside.*[16]

Figure 20 shows a complete list of all syntax restrictions inside of classes. Using one of the elements in Figure 20 inside a class will result in a syntax

[16] As of the Release 6.40 ABAP documentation, all obsolete language elements are either marked as obsolete or placed in a special chapter of obsolete statements that explains why the elements are obsolete and how to replace them with better concepts.

Figure 20 *Syntax Restrictions*

Context	Restrictions
Syntax notation	No special characters in names; no negative length specifications; no multi-line literals
Declarations	LIKE references to data objects only; no implicit lengths or decimal places in TYPES; no length specifications for data types i, f, d, or t; no operational statements in structure definitions; FIELDS, RANGES, INFOTYPES, TABLES, NODES, COMMON PART, OCCURS, NON-LOCAL not permitted
Operations	CLEAR … WITH NULL, PACK, MOVE ... PERCENTAGE, ADD-CORRESPONDING, DIVIDE-CORRESPONDING, SUBTRACT-CORRESPONDING, MULTIPLY-CORRESPONDING, ADD THEN ... UNTIL ..., ADD FROM ... TO ..., CONVERT {DATE|INVERTED DATE} not permitted

(continued on next page)

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 20 *(continued)*

Context	Restrictions
String processing	Not permitted on numeric data objects
Field symbols	No implicit types; FIELD-SYMBOLS … STRUCTURE, ASSIGN ... TYPE, ASSIGN LOCAL COPY OF, ASSIGN TABLE FIELD not permitted
Logic expressions	Operators ><, =<, => not permitted; table used with IN must be a selection table
Control structures	No operational statements between CASE and WHEN; ON-ENDON not permitted
Internal tables	No header lines; no implicit work areas; no redundant key specifications; compatible work areas required where necessary; obsolete READ variants, COLLECT ... SORTED BY, WRITE TO itab, PROVIDE (short form) not permitted
Procedures (methods)	No implicit type assignment; compatible initial values only; passing of sy-subrc and raising of undefined exceptions not permitted
Program calls	No joint use of USING and SKIP FIRST SCREEN when calling transactions; passing formal parameters implicitly in CALL DIALOG not permitted
Database accesses	No implicit work areas in Open SQL; READ, LOOP, REFRESH FROM on database tables not permitted; VERSION addition to DELETE and MODIFY not permitted; no PERFORMING addition in Native SQL
Data clusters	No implicit identifiers; no implicit parameter names; no implicit work areas; MAJOR-ID and MINOR-ID not permitted
Lists	DETAIL, SUMMARY, INPUT, MAXIMUM, MINIMUM, SUMMING, MARK, NEW-SECTION, and obsolete print parameters not permitted

error. You can find the recommended replacements in the ABAP documentation.

Let's look at some examples that demonstrate the benefits of the cleaner ABAP Objects syntax.

Figure 21 shows the obsolete definition of an internal table in procedural ABAP on the left, and how it must be done inside of a class in ABAP Objects on the right. The obsolete definition of internal tables in procedural ABAP is a significant source of confusion: In addition to the internal table, a second data object (the header line) is defined with the same name (here, *itab*). Since working with two different data objects that have the same name is pretty confusing, the syntax enforced in ABAP Objects defines just the internal table. Without the implicit definition of a header line, it is immediately obvious that the *CLEAR* statement initializes the internal table. In procedural ABAP, you always have to know if a statement refers to the table or to the header line, which often leads to programming errors (e.g., in Figure 21, it deletes the header line, not the table!). As a side effect, forbidding header lines in ABAP Objects forces you to use explicit work areas when working with internal tables, which further reduces the implicitness of your coding and leads to cleaner coding that is easier to understand.

Figure 22 compares a database access in procedural ABAP and how it is done in ABAP Objects. The short form of the *SELECT* statement, which can be used in procedural ABAP with a *TABLES* statement, is simply inappropriate for a 4GL language, which in almost all other cases is very narrative: The behavior of *TABLES* combined with *SELECT* is absolutely implicit. You have to know that *TABLES* implicitly defines a work area with the name of the database table (here, *dbtab*) and that all database statements

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 21 Defining an Internal Table in Procedural ABAP and ABAP Objects

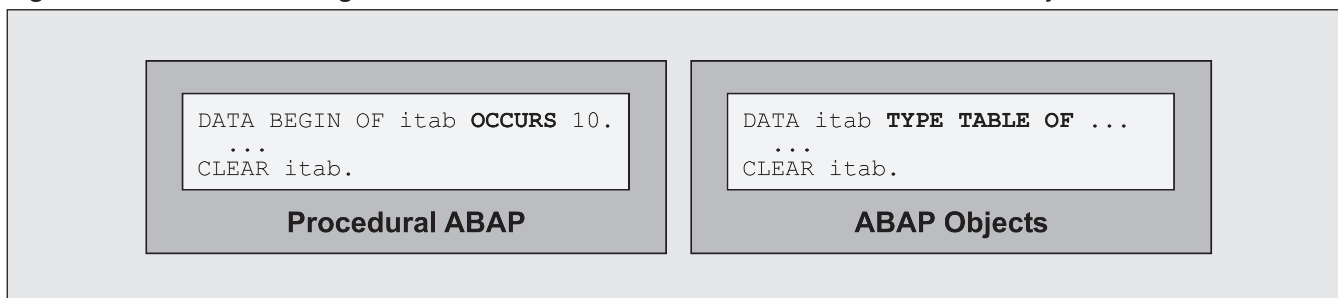


Figure 22 Accessing a Database in Procedural ABAP and ABAP Objects

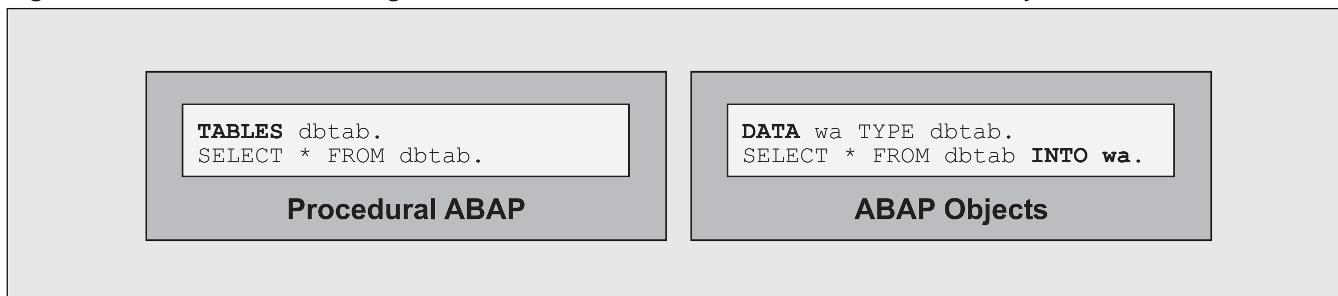
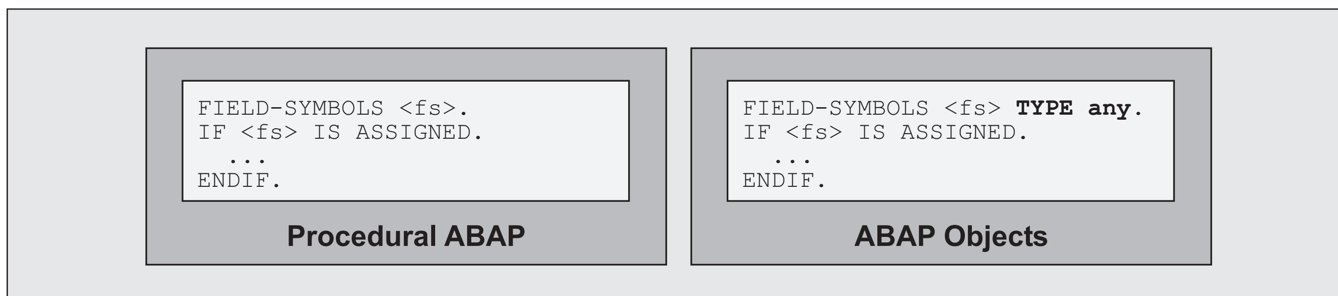


Figure 23 Explicit Typing in Procedural ABAP and ABAP Objects

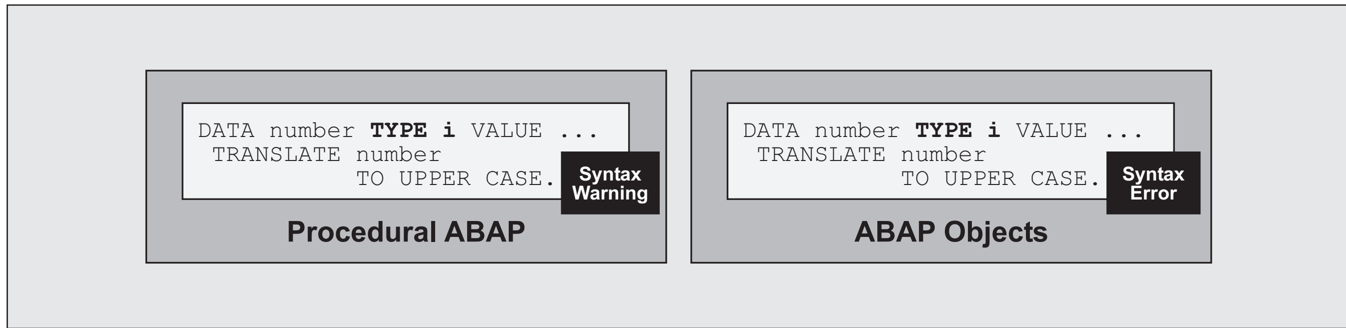


that work on the corresponding database table use this working area implicitly. So, in our example, *SELECT* is implicitly completed by the addition *INTO dbtab*. In ABAP Objects, this implicit behavior cannot occur. The *TABLES* statement is forbidden, and the syntax check forces you to use the explicit syntax variant that clearly expresses its semantics.

Figure 23 shows the benefits of another important improvement enforced in ABAP Objects, namely the explicit typing of formal parameters and of field symbols. Outside of ABAP Objects, you still can leave a formal parameter or a field symbol untyped. For field symbols, this can have unexpected effects: If a field symbol is not explicitly typed, for historical reasons the runtime environment implicitly assigns the predefined data object *space* to the field symbol. Therefore, the result of the logic expression in the procedural ABAP example in Figure 23 is true, which is certainly not what you would expect. Since in ABAP Objects every field symbol must be typed, you would declare it as being of type *any*, as shown in the ABAP Objects example in Figure 23, and the result of the logic expression would be false, as expected.

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 24 *Incorrect Data Handling in Procedural ABAP and ABAP Objects*



As a final example, **Figure 24** demonstrates how ABAP Objects prohibits incorrect data handling. Outside of classes, using procedural ABAP, you will get a syntax warning if you try to perform character operations on numerical fields — nothing more.[17] Inside of classes, using ABAP Objects, on the other hand, performing character operations on numerical fields is strictly forbidden. You will get either a syntax or a runtime error when you try to do so. The reason is that the result of such an operation is undefined in ABAP Objects and cannot be used. Working with ABAP Objects guarantees that operations on data are performed only if the operation is appropriate for the given type of data. (For interested readers, the sidebar on the next page shows how this aspect of cleaning up the language was driven even further with the introduction of Unicode-enabled programs.)

Reason #3: ABAP Objects Provides Access to New ABAP Technology

Since new ABAP-based SAP technology uses ABAP Objects, you will inevitably be confronted with it in your projects. A lot of basic functionality is already shipped in ABAP Objects classes, including:

- Frameworks for user dialogs, such as the SAP Control Framework (CFW), Business Server Pages (BSPs), and Desktop Office Integration (DOI), for example
- Frameworks for persisting data in the database (Object Services) and Shared Objects (area classes)
- Service classes, such as *CL_GUI_FRONTEND_SERVICES* for working with data at the presentation server, for example
- Language-related classes, such as Run Time Type Services (RTTS) classes or *CL_ABAP_EXPIMP* subclasses for extended *EXPORT*/*IMPORT* functionality, for example

This trend of implementing basic services in ABAP Objects classes instead of in functions or by creating new language constructs will continue, which means that if you want to work with new ABAP technology, you will have to use ABAP Objects at some point. In such cases, especially when you are starting new projects, where you have to use class-based frameworks anyway, we highly recommend switching to ABAP Objects completely. Although syntactically allowed, try to avoid mixing procedural and OO techniques within an application; otherwise, your applications will become more complex instead of simpler, because different programming models with different rules will be involved. Restrict your use of procedural programming concepts to where they are really needed:

☑ Use function modules only if they are really necessary — for encapsulating classical screens (Dynpros) in function pools (see the appendix) or offering functionality to be used remotely via Remote Function Calls (RFCs), for example.

[17] Please note that you will only get a syntax warning if you check the program explicitly. Otherwise — and this is where things get dangerous — the code will be activated and executed without any warning that there are errors.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Improved Semantics in Unicode Programs

As of Release 6.10, SAP Web Application Server (SAP Web AS) supports both Unicode and non-Unicode systems. In non-Unicode systems, characters are usually represented by one byte each, and many ABAP programming techniques depended on that fact. Unicode systems are based on Unicode character representation, where one character is represented by more than one byte. Therefore, before a system can be converted to Unicode, ABAP programs have to be modified wherever an explicit or implicit assumption is made about the internal length of a character.

To support this conversion, some new syntax and semantic rules were introduced with SAP Web AS Release 6.10. These rules are valid for programs where "Unicode checks active" is selected in the program attributes, which identifies the program as a Unicode program. In a Unicode system, all ABAP programs must be Unicode programs. In non-Unicode systems, an ABAP program can be either a Unicode or non-Unicode program.

Although optional, we strongly recommend setting the "Unicode checks active" attribute for all your programs and classes, even if you have no immediate plans to switch to Unicode. Just as many outdated, risky language constructions have been declared obsolete with ABAP Objects, the rules for Unicode enhance the quality and maintainability of programs. If you set the "Unicode checks active" attribute for an existing program and it requires major modifications, this is usually an indication that an error-prone and therefore questionable programming style was used to create it.

(continued on next page)

- ☑ Avoid the use of subroutines for internal modularization — their parameter interfaces are far less advanced than methods. Subroutine parameters are positional parameters, while method parameters are keyword parameters. Also, other than in methods (or function modules), subroutines do not prohibit write access to input parameters passed by reference. In addition, when calling subroutines, the type check for parameters is done only at runtime; for methods it is done at compile time. We recommend using static methods defined by *CLASS-METHODS* instead of subroutines. Even if you continue working in a procedural way (if you don't need multiple object instantiation in your application, for example), if you use static methods instead of subroutines, your programs will benefit from the advanced parameter interface and stricter syntax rules of methods.

Conclusion

Programming with ABAP Objects means programming in a modern style that exploits the benefits of a paradigm that was invented to solve the problems of complex software projects — object-orientation. After reading this article, you now know that ABAP Objects offers you the advantages of:

- ☑ Better encapsulation
- ☑ Support for multiple instantiation
- ☑ Better techniques for reusing code
- ☑ Better interfaces
- ☑ An explicit event concept

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

(continued from previous page)

The rules for Unicode programs are:

- ☑ Static type checks are specified more precisely.
- ☑ Byte and character strings are processed separately.
- ☑ Structures are handled appropriately according to their type, using structural equivalence rules.
- ☑ Uncontrolled memory manipulation is no longer permitted.

The following table shows the complete list of all syntax and semantic changes in Unicode programs:

Context	Restrictions
Offset/length accesses	Only performed on character type or byte type elementary fields, and for structures only on flat character type beginning parts
Memory accesses	No access to memory outside a data object
Byte/character processing	Clear separation between byte string and character string processing; explicit specification with IN BYTE MODE or IN CHARACTER MODE; appropriate types expected (for character strings this means only c, d, n, t, string, and flat structures with purely character type components)
Structures	When assigning and comparing, you must take the Unicode fragment view into consideration
File interface	Implicit opening of files no longer permitted; access, storage, and coding type must be specified explicitly; no write access to read-only files
Conversions	TRANSLATE ... CODE PAGE and TRANSLATE ... NUMBER FORMAT not permitted
Open SQL	Stricter conditions for work areas
Typing	Stricter checks on assignments to field symbols and formal parameters typed with STRUCTURE
Function modules	A formal parameter of a function module specified in the call must be available

And even if you have no plans to fully exploit the benefits of ABAP Objects by shifting to an object-oriented programming model, don't miss out on the opportunity to improve your procedural programming using ABAP Objects capabilities:

- ☑ Use methods as much as possible, even if you stay within the procedural programming model.
- ☑ Replace the use of subroutines with static methods.
- ☑ Use function modules only when technically necessary (for RFCs, screen encapsulation, etc.).
- ☑ Disentangle procedural ABAP from ABAP Objects.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

☑ Decouple screen programming from application programming.

☑ Never uncheck the "Unicode checks active" checkbox.

With the knowledge you've gained in these pages, it would be a good time for you to give serious thought to your current approach to ABAP development. We hope we've convinced you that programming with ABAP Objects does not mean adding a new level of complexity to an already-complex language, but rather a new level of opportunity and flexibility to fit your needs best:

- On the low end of the scale, you can use ABAP Objects without instantiating objects at all. As a programmer who is mainly busy with writing simple applications, such as small report programs, for example, you might simply start using static methods instead of subroutines, and check out how your programs are improved.

- On the high end of the scale, you might want to start a new, fully object-oriented project in order to exploit all the benefits the OO paradigm can offer. Such a project requires a detailed planning phase and a lot of OO modeling, which is far beyond the scope of this article.

For most developers, the ideal fit is likely somewhere in between these two extremes. And keep in mind that like that other famous object-oriented language, Java,[18] coding with ABAP Objects doesn't necessarily require the use of UML,[19] which means you need not undertake any extensive OO modeling efforts. So don't let a fear of the OO paradigm rob you of the benefits you and your applications can gain from the possibilities offered by ABAP Objects. Once you get the hang of it, you'll wonder why you ever programmed any other way!

---

[18] By the way, you can program in a Java style in ABAP Objects by creating classes with a main method that is linked to a transaction code.

[19] Unified Modeling Language, a tool that is widely used for object-oriented modeling.

Horst Keller studied and received his doctorate in physics at the Darmstadt University of Technology, Germany. After doing research in several international laboratories, in 1995 he joined SAP, where he worked in the ABAP Workbench Support group for three years. During that time, he authored the ABAP programming books for the printed ABAP Workbench Documentation Set for SAP R/3 Release 3.0. Horst is currently a member of the NetWeaver Development Tools ABAP group, where he is responsible for information rollout. He is documenting the ABAP language with an emphasis on ABAP Objects. He is the author of several articles and workshops about this subject, and he is the main author of the books "ABAP Objects" and "ABAP Reference." Besides his writing tasks, Horst is developing the programs that display the ABAP documentation and corresponding examples in the SAP Basis system. He can be reached at horst.keller@sap.com.

Gerd Kluger studied computer science at the University of Kaiserslautern, Germany. After receiving his degree, he worked for a company whose main focus was the development of programming languages for business applications. He was responsible for the development of the compiler and programming environment for the object-oriented programming language Eiffel. Gerd joined SAP in 1998, and since then has been working in the NetWeaver Development Tools ABAP group. His main responsibility is in the development of ABAP Objects, the new class-based exception concept, and the further development of system interfaces, especially with regard to the file system. He can be reached at gerd.kluger@sap.com.

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Appendix: Decoupling Classical Screen Programming from Application Programming Using ABAP Objects

As you are likely well aware, application and screen programming are closely intermingled in many existing ABAP programs.

Such programs are driven by user inputs on screens, and the application logic is coded in dialog modules or event blocks, which are part of the classical screen-based ABAP programming model (dialog programming and reporting). ABAP Objects, however, was mainly designed as a language for application programming, and there is no object-oriented support for the classical screen programming techniques (Dynpros, selection screens, messages, and lists).[1]

If you want to reorganize your ABAP coding according to the OO paradigm, or write new coding in ABAP Objects, the (justified) question is: How can I use ABAP Objects when it is not supported by the classical screen programming integrated into my ABAP program?

[1] A class pool cannot have screens as components. ABAP program types that support screens are module pools, executable programs, and function pools.

The answer is: Decouple the application model from the user interface! Using an example application program that works with Dynpros, here we will show you how to decouple (classical) screen programming from application programming using ABAP Objects.[2]

Figure 1 shows how a function pool can encapsulate all the screens needed by an application. In the example shown, the function pool contains a selection screen *100* and a general screen *200*. The data interface between the function pool and general screen is declared with the *TABLES* statement.[3] The function modules handle the screens and offer the outside user parameter interfaces, to send data to and receive data from the screens.

If internal modularization is needed inside such

[2] Another example is the function module *REUSE_ALV_GRID_DISPLAY*, which displays lists in an ALV grid and encapsulates the complete screen handling.

[3] Remember that the *TABLES* statement is forbidden in ABAP Objects (because of an inadequate data interface and implicit behavior), but that it is needed for data transfer between classical screens and ABAP programs.

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 1 ***Encapsulating Screens in a Function Pool***

a function pool, it should be done via local classes. If one of the function pool's screens contains GUI controls, the respective event handling should also take place in local classes of the function pool. In the simple case shown in Figure 1, no dialog modules are needed.

If you want to use automatic mechanisms for input checks (statements *FIELD* and *CHAIN* of the screen flow logic), you have to program those modules inside the function pool as well. Remember in such cases to put as few lines of coding into the modules as possible and call local methods instead. At least, for the sake of data encapsulation, avoid working with global data directly.

Figure 2 (on the next page) shows how a class *flights* might use the screens of the function module *get_plane_type*. Each object of the class represents a flight from the database table *sflight*. When creating an object, the constructor calls the function module that handles the selection screen for the business keys of the flight. A method *change_plane_type* calls the function module that handles the screen that asks for the plane type (*get_plane_type*), but there is no screen programming inside the class.

A program that only (or at least widely) uses ABAP Objects, and that decouples the application model from the user interfaces, is well prepared for future technology changes in the presentation layer

©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 2 *Using the Screens of the Function Module in a Class*

(e.g., SAPGUI or Business Server Pages), since application coding and screen programming is no longer intermingled, and a change to a new presentation technique will be restricted to a change of some procedure calls. It is interesting to note that such a separation is also possible in procedural ABAP, but it is not enforced there, unlike with ABAP Objects. The simple fact that screens are not allowed as components of classes in ABAP Objects will force you to decouple them from your future application logic written in ABAP Objects, so now is as good a time as any to get comfortable with the process.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

SAP Professional Journal
www.SAPpro.com

the leading technical journal for SAP administrators, architects, consultants, and developers

About SAP Professional Journal

Elevate your mastery of SAP technology with in-depth tutorials and expert guidance in *SAP Professional Journal*. Top independent consultants, experienced SAP customers and the technical gurus who write the code for SAP contribute proven best practices and their favorites tips and techniques for developers, administrators and specialists in SAP technology. Articles cover everything from ABAP to Java programming, from performance optimization to integration.

Each issue features step-by-step instruction on the SAP projects that top your to-do list right now. You can immediately put to use practical advice that can't be found anywhere else.

Learn Directly From the Experts

All *SAP Professional Journal* authors are among the best and brightest minds in the SAP community. The unique articles they write are training courses unto themselves. Articles for developers often feature code samples and step-by-step development techniques, while articles for administrators feature detailed screenshots and in-depth guidance to enhance performance. Typical articles span 20-30 pages, and overall coverage blends operational and theoretical background with day-to-day implications of how and why SAP technology operates as it does, and how best to incorporate that technology into your own IT environment. In addition we provide a complete download library of all source and sample code featured in our articles.

Published bimonthly, every issue features highly-specific articles, each focused on a narrow technical area or particular SAP offering. Every author is uniquely qualified to take advanced, complex concepts and present them in an easy-to-understand format, at a level of technical detail available nowhere else, to help advance your understanding of SAP technology.

Invest in Your Team

Keeping up with advances in SAP technology is a constant endeavor that is critical to your success, as well as to achieving your enterprise's key business goals. Now your whole team can do so with access to every single article ever published in *SAP Professional Journal* through our electronic license program. This program is the most affordable way to access the *SAP Professional Journal* online archive – your whole team gets one-click access to new articles, plus the ability to search instantly through more than 3,500 pages of accumulated solutions, for a fraction of the cost of multiple subscriptions.

Your team is your strongest asset, and giving them access to *SAP Professional Journal* is the best investment you can make in them -- guaranteeing your organization's SAP knowledge and skills stay on the cutting edge. To learn more about our Electronic License Program call 781-751-8799 or send a message to licenses@sappro.com.

Subscribe Risk-Free

SAP Professional Journal is backed by a 100% money-back guarantee. If at any point, for any reason, you are not completely satisfied, we will refund your subscription payment IN FULL. Don't wait any longer to benefit from the most respected technical journal for SAP expertise. This is the most in-depth publication written specifically for SAP professionals like you. Subscribe to *SAP Professional Journal* today at www.SAPpro.com or call us at 781-751-8799.

SAP Professional Journal 990 Washington Street, Ste. 308 Dedham MA 02026, USA

www.SAPpro.com