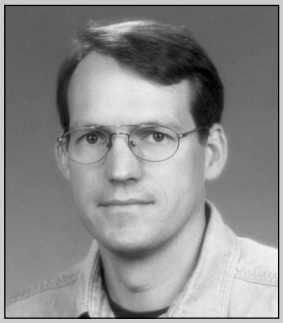
ABAP Programming — An Integrated Overview

Horst Keller



Horst Keller is a member of the ABAP language group, where he is responsible for information rollout. He is documenting the ABAP language with an emphasis on ABAP Objects, and develops and teaches classes on ABAP programming. Horst is also writing presentation programs for the ABAP documentation and the corresponding examples in the R/3 Basis system.

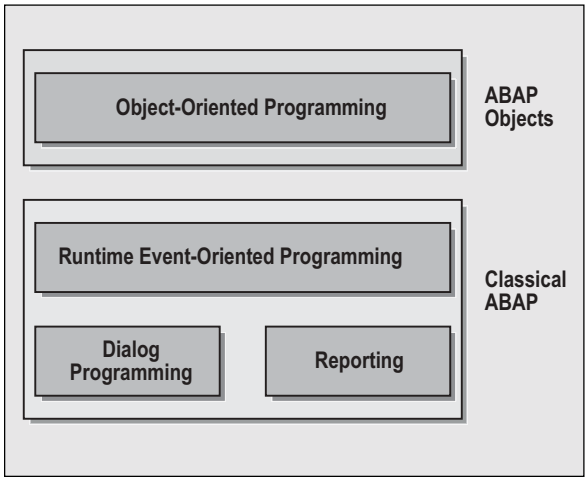
(complete bio appears on page 109)

ABAP has undergone a profound evolution. When it first came on the scene, it was a mainframe-based macro assembler language used exclusively for reporting purposes. Nowadays, ABAP is a hybrid language, as shown in **Figure 1**, that encompasses two programming models.

In the classical ABAP programming model, program execution is based on events that are triggered by the ABAP runtime environment. The two classical types of application programs, reporting and dialog programs, are governed by this programming model. In classical reporting you know these runtime events as `START-OF-SELECTION`, `GET`, and so on. In classical dialog programming, the most important runtime events are `PROCESS BEFORE OUTPUT` (PBO) and `PROCESS AFTER INPUT` (PAI).

The second ABAP programming model, added just recently, is object-oriented programming. In a pure object-oriented world, the program execution is based on the instantiation of classes and on method calls, but in ABAP, objects can also be used in classical programs.

Figure 1



Just as the language has changed, so must our usage and view of the language. Most ABAP programmers are mired in the classical ABAP programming model, building applications that revolve around reports *or* dialogs. There tends not to be a lot of crossover between these two development camps. Before Release 4.0, even SAP's documentation and training taught these topics separately, treating the interconnection between the two — for example, calling a dialog screen during list processing — as a special feature. As a result, you find report programmers who don't know the meaning of a module pool, and application specialists who know how to program advanced transactions, but can't wield a `START-OF-SELECTION` statement.

With Release 4.0 and the dawn of ABAP Objects, we're replacing the classical distinction between reporting and transaction programming with an integrated view in documentation and training, recognizing the simple fact that all application logic is programmed in ABAP and that the application can communicate with the user via screens and with the database via a common interface. Changing between programming types (i.e., doing reporting during a transaction) is, and technically always has been, nothing special. This is a natural feature of the language — one that should be exploited if you want to use ABAP efficiently.

The aim of this article is to provide new ABAP programmers with a fresh, integrated overview of the language, and to provide experienced ABAP programmers, who may be used to reporting *or* dialog programming, with insights to overcome the gap between these classical programming types and ultimately better prepare you for ABAP Objects.

A Simple ABAP Program

Let's start our discussion with the very simplest of ABAP programs — the kind you learn at the start of a beginners' ABAP class, and which is shown in **Figure 2**.

This is about as elementary as it gets. The `REPORT` statement opens the program `select_and_write`. The `DATA` statement declares data named `wa_splfi`. The `SELECT` statement reads data from a database table `spfli` into `wa_splfi`. The `WRITE` statement writes data to the screen.

The thing worth noting is that even this short and simple program showcases the most important features that are common to nearly every ABAP program: it declares data, and it communicates with the database as well as with a screen.

Now consider that an R/3 system has a complex, multi-tiered architecture with at least three software layers, namely the presentation layer, the application layer, and the database layer. (Other layers, such as the Internet-enabling layer, are extensions to the basic three-tiered architecture.)

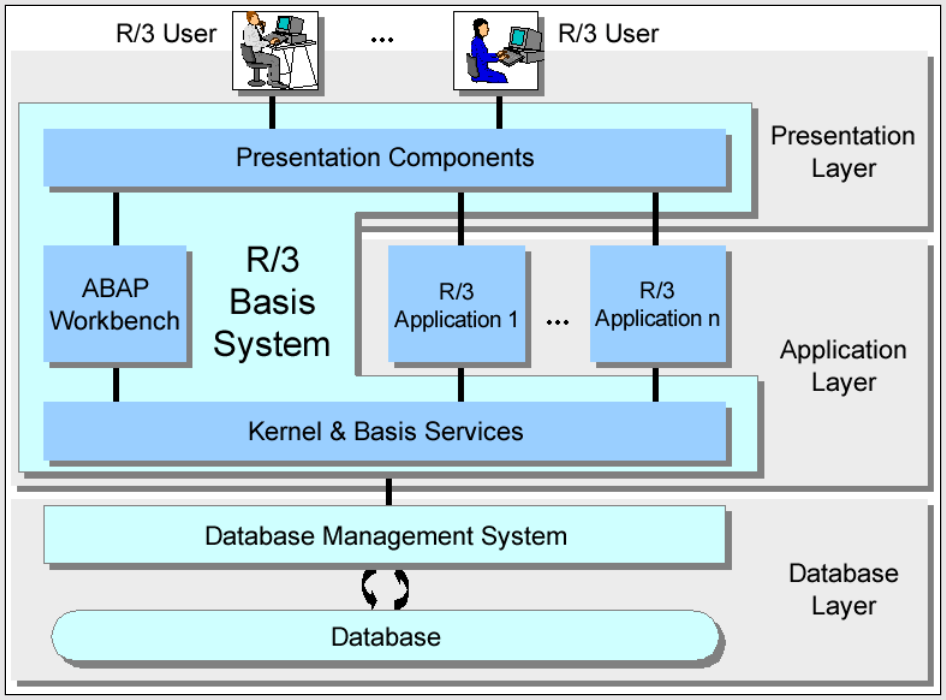
The connection between the layers depends on the hardware configuration of the system, but is generally realized by network links. So the four statements in this short, simple coding exercise are actually a complete example of client/server computing: the program is executed on an application server; data is transferred from the database server to the program; a screen with a user interface is defined and sent to the presentation server.

Don't lose sight of this when writing ABAP programs, especially when it comes to database accesses, where the load on the network plays an important role for the performance of a program. Look closely at the code again. You see that all data (*****) of one database line is selected, but only three fields are displayed. Already, in this simple program, the network load can be reduced by replacing the ***** with a list of field names. Of course such measures are more important when you select more than one line.

Figure 2 A Very Simple ABAP Program

```
REPORT select_and_write.
DATA wa_spfli TYPE spfli.
SELECT SINGLE * FROM spfli
               INTO wa_spfli
               WHERE carrid = 'UA' AND connid = '0941'.
WRITE: wa_spfli-airpfrom, wa_spfli-airpto, wa_spfli-fltime.
```

Figure 3 Logical Components of the R/3 Basis System



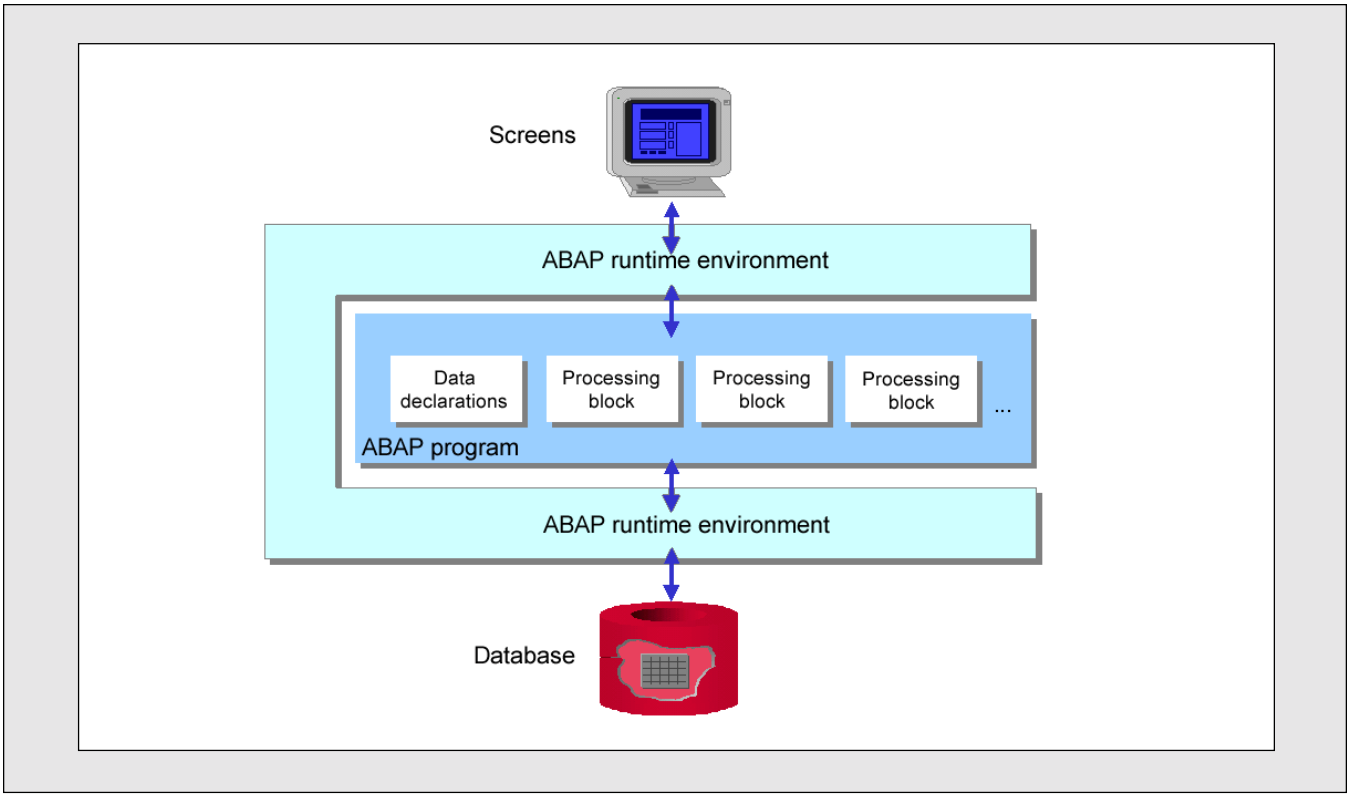
The Runtime Environment of ABAP Programs

The runtime environment has a direct impact on the execution of ABAP programs and is the basis for the classical ABAP programming model. Anyone who wants to advance beyond simple programming needs to understand it and how it enables client/server computing within an R/3 environment.

The central platform for all ABAP applications is the R/3 Basis system. **Figure 3** shows the three logical components of the R/3 Basis system and how they map to the R/3 multi-tiered architecture:

- The presentation components are responsible for the interaction between the R/3 system and the user.
- The ABAP Workbench component is a full-fledged development environment for applications in the ABAP language. It is fully integrated in the R/3 Basis system, and, like other R/3 applications, is itself written in ABAP.

Figure 4 The ABAP Runtime Environment



- The kernel and Basis services component provides a hardware-, operating system-, and database-independent runtime environment for ABAP programs (applications). This runtime environment is written mainly in C and C++. All ABAP programs run on virtual machines (VM) within this component.

All R/3 application programs are embedded in the R/3 Basis system. This makes them independent of the hardware and operating system that you are using. However, it also means that you cannot run them outside the R/3 system. In Figure 3, you see that the communication of the ABAP application programs with the screens of the presentation layer and the database system of the database layer is not direct. These communications are mediated by the R/3 Basis system. We simplify this picture in **Figure 4** by saying that each ABAP program is encapsulated in a runtime environment delivered by the R/3 Basis system.

In addition to mediating the dialogs between ABAP programs and screens or databases, the runtime environment is also responsible for the execution of an ABAP program. Figure 4 also shows the structure of ABAP programs. Each ABAP program consists of a global data declaration part and self-contained processing blocks. Apart from its global data declarations, any statement of an ABAP program is definitely part of exactly one processing block. Processing blocks cannot be nested. The ABAP runtime environment triggers the execution of the processing blocks, and therefore steers the time sequence in which ABAP programs are executed. The order of the processing blocks does not influence the program execution. When a processing block is triggered, its coding is processed sequentially. There are processing blocks that are triggered from the runtime environment and those that can be called with ABAP statements from other processing blocks. When the execution of a processing block is finished, control is given back to the caller.

Processing Blocks

Most of you know that all ABAP programs are made up of processing blocks. If you are familiar with classical reporting, you know the processing blocks that are defined by the statements `START-OF-SELECTION`, `GET`, and so on. If you do dialog programming, you are well-acquainted with the dialog modules defined between the `MODULE` and `ENDMODULE` statements. Other processing blocks, such as subroutines or function modules, are common to both classical programming styles. ABAP Objects does not change this organization of ABAP programs, but merely adds new types of processing blocks, such as methods, for example.

When you look at the statements of an ABAP program, you should always be able to recognize its processing blocks. Only with this knowledge are you able to understand the program flow.

The following sections describe the different processing blocks and how they are executed in different kinds of programs. Note that the ABAP Debugger supports you by showing the structure of a program and its processing blocks when you choose the function *Overview*.

Event Blocks

Event blocks are introduced by one of the event keywords listed in **Figure 5**. There is no explicit statement to end an event block. It is concluded by the next processing block. To make an ABAP program more readable, it is always a good idea to conclude an event block with a comment line! Event blocks have no local data area and no parameter interface. Their execution is triggered by events in the ABAP runtime environment.

Dialog Modules

Dialog modules are defined between the `MODULE` and `ENDMODULE` statements. They have no local data area and no parameter interface. They are called

Figure 5 Event Keywords

Program constructor event: `LOAD-OF-PROGRAM`

This event is triggered exactly once, when an ABAP program is loaded into the memory. The statements in this block can initialize the data of the program.

Reporting events: `INITIALIZATION`, `START-OF-SELECTION`, `GET`, `END-OF-SELECTION`

These events are triggered by the ABAP runtime environment while an executable program is running. `START-OF-SELECTION` is the standard processing block of an executable ABAP program. Each procedural statement in an executable program automatically belongs to `START-OF-SELECTION` if you do not define other processing blocks explicitly. The statements in the other blocks serve as application logic for report programs.

Selection screen events: `AT SELECTION-SCREEN` ...

This is a set of events that are triggered by the ABAP runtime environment when a selection screen is processed. The statements in these blocks can prepare the screen or process the user's input.

List events: `TOP-OF-PAGE`, `END-OF-PAGE`, `AT LINE-SELECTION`, `AT USER-COMMAND`

These events are triggered by the ABAP runtime environment while a list is being created or when a user performs an action on a list. The statement in these blocks can format the list or process the user's requests.

Screen events: `PROCESS BEFORE OUTPUT`, `PROCESS AFTER INPUT`

These events, abbreviated by PBO and PAI, are triggered by the ABAP runtime environment before a screen is sent and after a user action on a screen. The respective event blocks are not defined in the ABAP program but in the screen's flow logic.

during the PBO and PAI events from the screen flow logic of the ABAP program's screens, and they contain the application logic for the screens.

Classes

Classes are defined between the `CLASS` and `ENDCLASS` statements. The definition of a class consists of a declaration part, where its components are declared, and an implementation part, where its methods are implemented. Classes cannot be executed but serve as patterns for objects.

Procedures

ABAP supports three types of procedures. They have a local data area and a parameter interface. Procedures can be called internally from the same ABAP program or externally from another ABAP program. In the latter case, the procedure's ABAP program is loaded into the memory, in addition to the calling ABAP program.

We differentiate between:

- **Subroutines**

  Subroutines are defined between the `FORM` and `ENDFORM` statements in any ABAP program except class pools. They can have positional parameters and are called with the `PERFORM` statement.

- **Function modules**

  Function modules are defined between the `FUNCTION` and `ENDFUNCTION` statements in programs of type function pool. They can have keyword parameters and are called with `CALL FUNCTION`.

- **Methods**

  Methods are defined between the `METHOD` and `ENDMETHOD` statements in the implementation part of classes. They can have keyword parameters and are called with `CALL METHOD` or triggered by events of ABAP Objects. Functional methods that have exactly one resulting parameter can be used as operands in expressions. We distinguish between instance and static methods. While instance methods are bound to the objects of a class, static methods can always be called.

Looking back at our simple code example in Figure 2, you see that there is no explicit processing block defined. This means that all procedural statements implicitly belong to the standard event block `START-OF-SELECTION`. **Figure 6** shows a complete version of the program.

Program Execution

At this point, even for the new ABAP programmers among us, it should be clear that an ABAP program is a container for processing blocks that are either triggered by the runtime environment or called by ABAP statements from other processing blocks. When a program is executed, at least one processing block must be triggered from outside the program by the runtime system. Therefore, starting an ABAP program actually means to load it into memory and to start a process in the runtime environment that triggers the program's processing blocks. The kind of process that is started in the runtime system and the sequence in which it triggers the processing blocks is totally independent of their sequence in the program. It depends solely on the ABAP program's type.

You enter a program's type in its attributes when you create a program. There are two types of programs that can be started by the user: executable programs and module pools. Historically, executable programs are used for reporting, and module pools are used for dialog programming. Report programmers are therefore familiar with executables, and dialog programmers are familiar with module pools. Let us look now at the technical difference between them. This will help you to decide which type to choose when you create a new program.

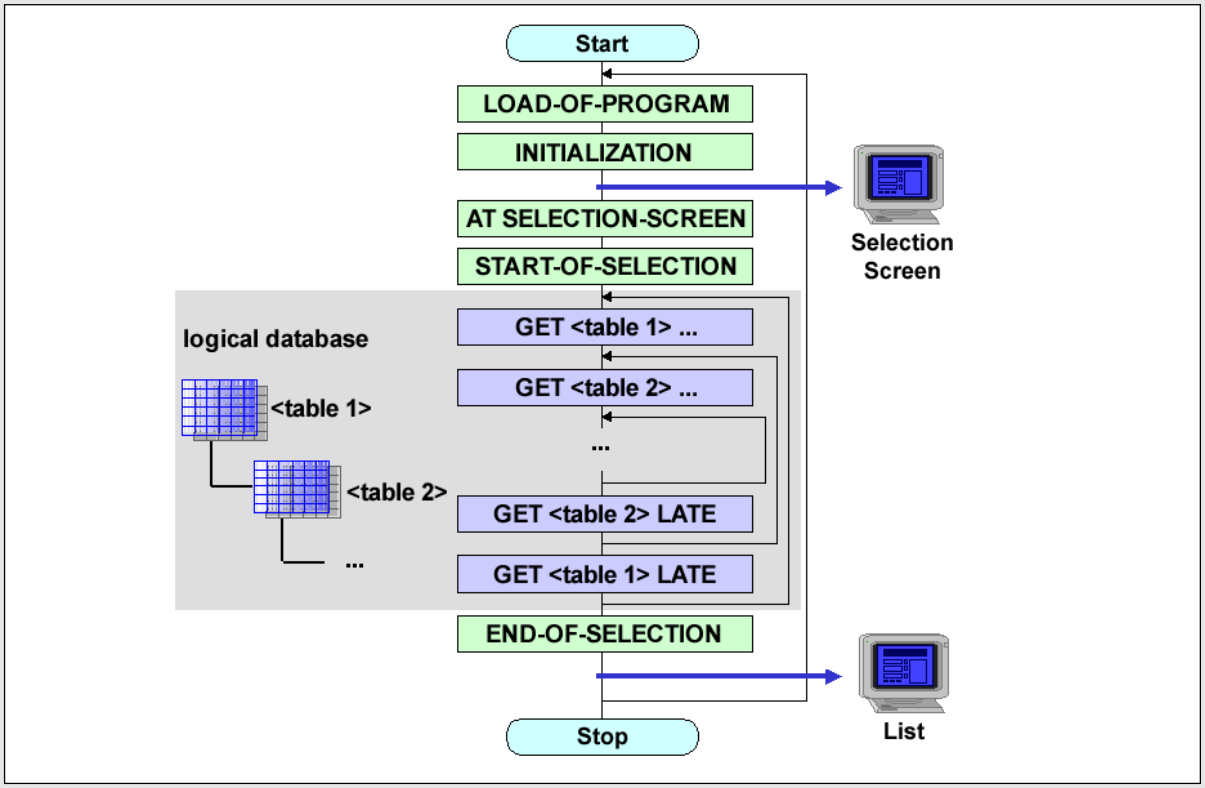
Executable Programs (Type 1)

An executable program is introduced with the

Figure 6 A Second Look at Our Simple ABAP Program

```
REPORT select_and_write.
************************************************************************
* Global data declarations
************************************************************************
DATA wa_spfli TYPE spfli.
************************************************************************
* Processing blocks
************************************************************************
START-OF-SELECTION.
  SELECT SINGLE * FROM spfli
                  INTO wa_spfli
                  WHERE carrid = 'UA' AND connid = '0941'.
  WRITE: wa_spfli-airpfrom, wa_spfli-airpto, wa_spfli-fltime.
************************************************************************
```

Figure 7 The Runtime Sequence of Events



REPORT statement. You can start it by simply typing its name in an appropriate R/3 screen, for example after choosing System → Services → Reporting. By starting an executable, you start a special process in the ABAP runtime environment that triggers a given sequence of events. If the program contains event blocks for some or all of those events, they are executed in exactly that sequence. If there are no event blocks defined for some events, the respective events are simply disregarded. **Figure 7** shows the sequence

of events triggered by the runtime environment when executing an executable program.

As for all ABAP programs, the program constructor LOAD-OF-PROGRAM is triggered first. Then, a reporting-specific sequence is processed. If there is a standard selection screen defined in the program, this screen is automatically displayed after the event INITIALIZATION. In the respective event block, the selection screen can be initialized. User actions on the selection screen trigger AT SELECTION-SCREEN events, where the user's input can be processed. Then, the standard event START-OF-SELECTION is triggered.

The following GET events are triggered only if the executable program is connected to a logical database. Logical databases are special ABAP programs that contain the definition of a selection screen and a set of special subroutines for reading data from database tables. During the execution of an executable program with a logical database, the selection screen of the logical database is displayed and its subroutines are called by the runtime environment in a sequence that results from the logical database's hierarchical structure. When a subroutine in the logical database has selected a line from the database, the runtime system triggers the respective GET event and the executable program can process the data that is delivered in a common data area. Logical databases are reusable components that hide the details of the data selection from the main ABAP program.

Note that from Release 4.5A on, it is also possible to call logical databases from any ABAP program, and that we plan to encapsulate them in classes in a coming release. The latter measure will clearly distinguish their functions and data from their ABAP program's clients.

When the logical database has finished its data selection or directly after START-OF-SELECTION, if there is no logical database connected, the runtime system triggers END-OF-SELECTION. In the respective event block, all data that is selected by the logical database can be processed — e.g., in an internal table. Without a logical database, an event block for END-OF-SELECTION does not make much sense, because the statements can also be coded in the event block for START-OF-SELECTION.

Finally, the runtime system automatically displays the list that was defined in these event blocks. When the user leaves the list, the program is either stopped, or, if a selection screen was displayed, the execution of the program is automatically started again. In the latter case, the program is only stopped when the user leaves the selection screen.

From Figure 7 you can now fully understand the execution of the program in Figure 6. It is an executable program and START-OF-SELECTION is the single event block of that program. There is no selection screen defined, but a list is written with the WRITE statement. After executing the event block, the list is displayed by the runtime environment.

The program flow as shown in Figure 7 supports classical reporting. But since you do not need to program event blocks for all possible events, you can use executable programs for all other types of application programs, for example, by calling screen sequences during START-OF-SELECTION.

Module Pools (Type M)

A module pool is introduced with the PROGRAM statement. You cannot start it by simply typing its name. You must use a transaction code. Defining a transaction code for a program simply means to link a screen of the program to this transaction code. Therefore, in order to be executable, a module pool must have at least one screen. Transaction codes are usually connected to menu entries in the R/3 user interface. A user can start a program via a transaction code by choosing such a menu entry or by typing the code directly into the OK field of the standard toolbar of any R/3 screen.

When a program is started via a transaction code, it is loaded into memory, and the runtime

environment triggers the `PROCESS BEFORE OUTPUT` (PBO) event of the screen that is connected to the transaction code. Then, the program flow is controlled by a sequence of screens that usually depends on the user's actions.

A module pool contains the dialog modules that are called during the PBO and `PROCESS AFTER INPUT` (PAI) events of its screens. Therefore, the program type "module pool" is appropriate for ABAP programs that are mainly dialog-driven. Nevertheless, module pools may contain not only dialog modules, but also all the event blocks for those events that can be triggered during their execution. While reporting events never occur during the execution of a module pool, the events for selection screen and list processing can be triggered, if such screens are defined in the program.

Module pools can be started via transaction codes only. Therefore, they must contain screens and dialog modules for those screens. But note that transaction codes and dialog modules are not restricted to module pools — they can be defined for all programs that have their own screens. So executable programs can be handled exactly as module pools — they just do so without the support of reporting.

A module pool contains the dialog modules that are called during the PBO and PAI events of its screens. Therefore, the program type "module pool" is appropriate for ABAP programs that are mainly dialog-driven. Nevertheless, module pools may contain not only dialog modules, but also all the event blocks for those events that can be triggered during their execution.

Program Modularization

Since ABAP programs must provide processing blocks in order to be executed and to react on the triggers from the runtime environment, they are automatically modularized. Besides this kind of modularization, you can use a special class of processing blocks, namely *procedures*, for a program-driven modularization and for code reuse. In classical ABAP, you are probably familiar with subroutines and function modules. ABAP Objects adds *methods* as new kinds of procedures. Principally, we can distinguish between internal and external modularization.

Internal Modularization

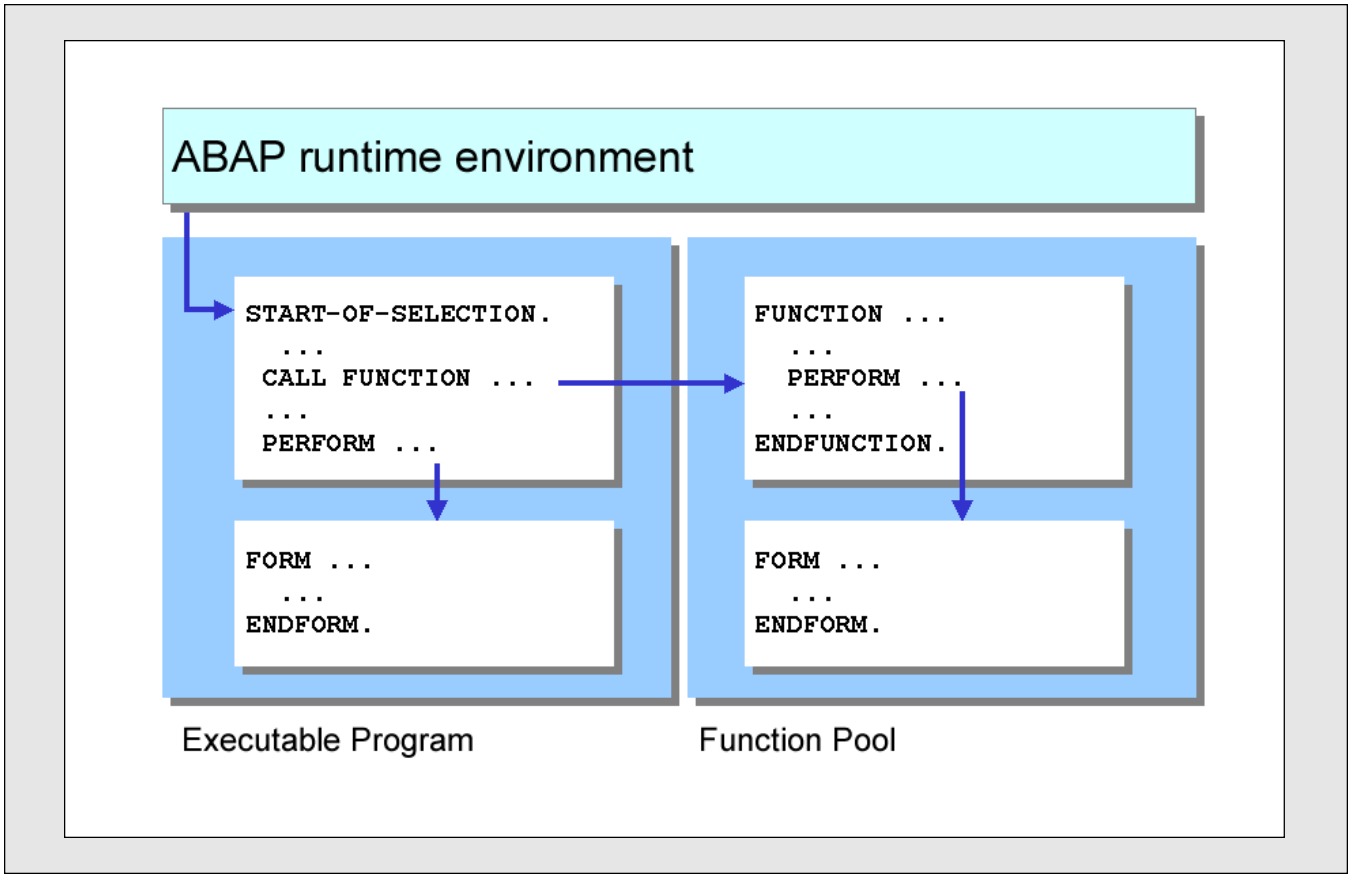
If you want to reuse sequences of code and to encapsulate functions and data within your ABAP program, you can modularize your program either with local subroutines or methods of local classes. After defining the procedures in your program, you can call them with the respective statements. You can pass data to and from these procedures. They both support a local data area.

External Modularization

If you want to create procedures that are accessible not just inside one program, but for all ABAP programs of the R/3 system, you can create function modules or methods in global classes. Function modules and global classes cannot be defined in executable programs or module pools. Therefore, you must create special types of ABAP programs called function pools (Type F) and class pools (Type K). Function pools and class pools cannot be created by simply choosing the program type in the ABAP Editor. You must use the tools Function Builder and Class Builder of the ABAP Workbench. These tools create ABAP programs of the respective type and structure and support you in defining the parameter interfaces of the procedures.

You define the function of such procedures by typing its procedural statements inside processing blocks that are generated by the tools. One function pool can contain several function modules and one class pool can contain one global class with several methods. Function pools can be modularized internally by local subroutines or methods of local classes.

Figure 8 Program Modularization in Classical ABAP Programming Model



Class pools can be modularized internally by local classes only. Function pools may contain, and therefore encapsulate screens. Class pools do not support screens yet. In a future release, the introduction of a package concept is planned, to restrict the access to global classes to members of the same package.

Figure 8 gives an overview of the different types of program modularization in classical ABAP programming. The ABAP runtime environment triggers `START-OF-SELECTION` in an executable program. There, a function module is called. The function pool of that function module is loaded into the memory if it is not already there. The function module itself makes an internal subroutine call. Then, the control returns to the executable program, and an internal subroutine is called there. The function pool stays in memory as long as the executable program is running.

Source Code Modularization

There is another kind of modularization that you should never confuse with real program modularization. This is source code modularization with Include programs (Type I). Include programs are not separately compiled units. Their source code is inserted into the programs in which a corresponding `INCLUDE` statement occurs. Include programs allow you to manage complex programs in an orderly way. For example, function pools and module pools use Include programs to store parts of the program that belong together. The ABAP Workbench sup-

ports you extensively when you create such complex programs by creating the Include programs automatically and by assigning them unique names. A special Include is the TOP Include of a program. If you name it by post-fixing TOP behind the program's name, it is always included in program navigation and in the syntax check. But you should never forget that an ABAP program always has the structure shown back in Figure 4, although it may be composed of many, even deeply, nested Include programs. It is a good exercise to open a function module with the Function Builder and to display the respective function pool. Now try to match the coding of the function pool with the structure of Figure 4.

There is another kind of modularization that you should never confuse with real program modularization. This is source code modularization with Include programs (Type I). Include programs are not separately compiled units. Their source code is inserted into the programs in which a corresponding INCLUDE statement occurs. Include programs allow you to manage complex programs in an orderly way.

Calling ABAP Programs

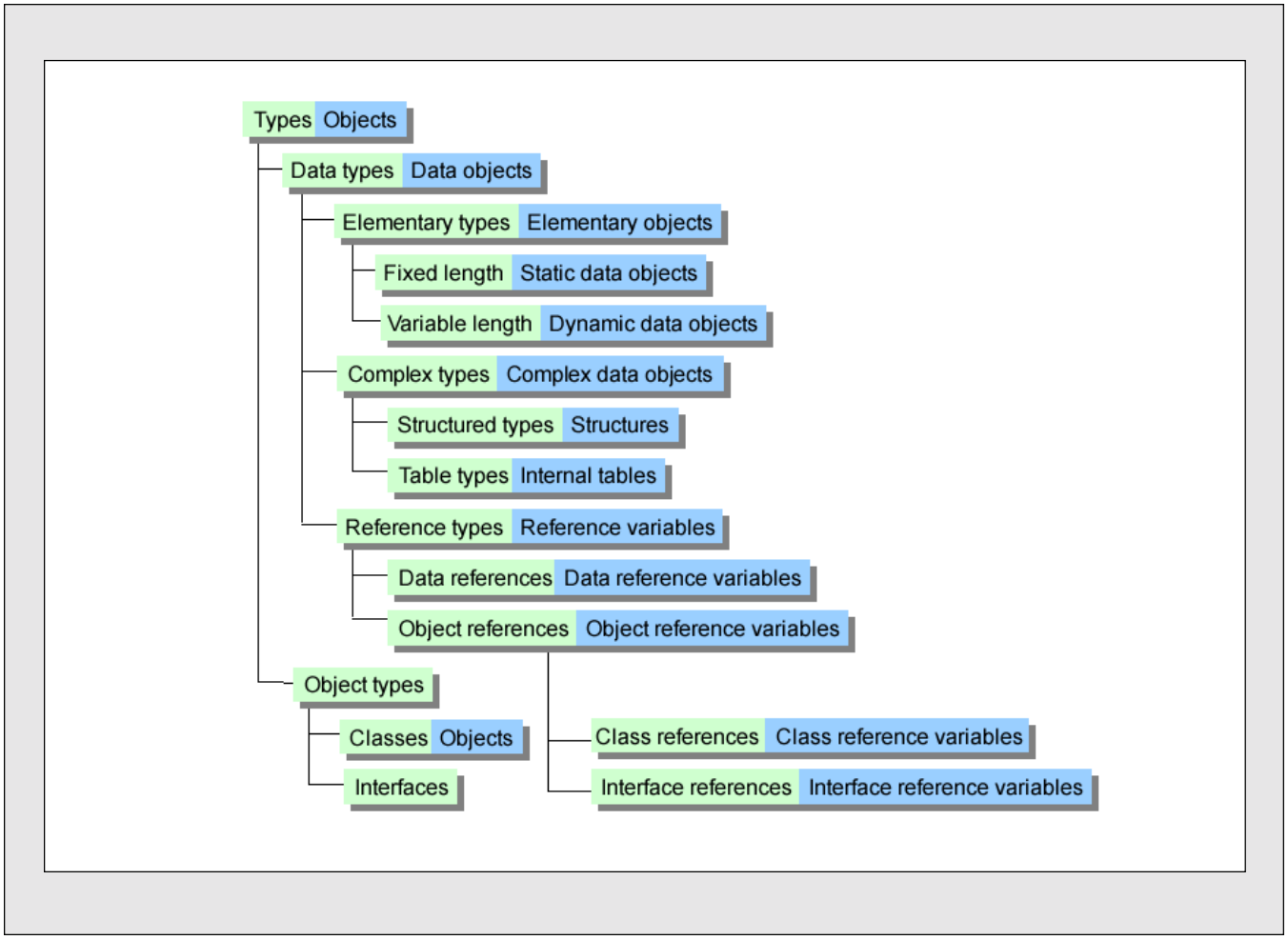
In addition to calling procedures from external programs, you can also start the execution of executable programs or module pools from other ABAP programs, either with the `SUBMIT` statement or with `CALL TRANSACTION`. The called programs are loaded into memory and executed by the runtime environment. With the `SUBMIT` statement, you start exactly the same process in the runtime environment that you saw in Figure 7. With `CALL TRANSACTION`, the runtime environment triggers the PBO event of the screen that is connected to the transaction code named in the statement. You can decide whether you want to exit the calling program or whether the control should return to the calling program after executing the called programs. It is interesting to note that executable programs are always started via a `SUBMIT` statement. When a user enters the program name in a transaction like SA38, he or she is already working with a running ABAP program, and when he or she chooses *Execute*, a `SUBMIT` statement is executed somewhere in a dialog module of the transaction. From a technical point of view the principal characteristic of an executable program is that it can be called by `SUBMIT`. Its principal characteristic from a user's point of view is that it can be executed by typing its name.

About ABAP Types and Data

It is clear that the main task of ABAP as a business application language is to handle and process data. Now that we know how ABAP programs are structured and executed, it is time to look at how various types of data are realized in ABAP.

In ABAP, data appears as the contents of data objects (fields). The prototype statement to define data objects is `DATA`. Each data object has appropriate memory assigned to it and a data type that describes its technical attributes. Data types can be mere attributes of data objects, but you can also define data types using the `TYPES` statement. Then, data objects can be viewed as instances of such data types. This concept of standalone data types was introduced with Release 3.0. Before that, you knew data types only as attributes of data objects. Since Release 4.5, the ABAP "type" concept is also mirrored in the ABAP Dictionary. Prior to Release 4.5, you used the ABAP Dictionary mainly as a tool for creating and maintaining database tables, and you could create ABAP data objects that referred to the structure of such tables. Besides, the only data types you could create in the ABAP Dictionary were so-called structures, which behaved like database tables without contents. Nowadays, you can use the ABAP Dictionary as an unrestricted tool and repository for global data types that are part of the "ABAP type universe."

Figure 9 *The ABAP Type Universe*



The ABAP Type Universe

Data types and objects are part of the ABAP type universe, which is shown in **Figure 9**.

Types are descriptions that do not occupy memory. Objects are instances of types, and do occupy their own memory space. A type describes the technical attributes of all of the objects with that type. Data types describe data objects. Object types are used in ABAP Objects.

All types of the ABAP type universe can either be defined locally, inside an ABAP program using the `TYPES`, `CLASS`, or `INTERFACE` statements, or globally, in the R/3 repository, using tools of the ABAP Workbench. Global data types are created in the ABAP Dictionary, and global object types are created with the class builder in the class library. While the types in the R/3 repository are visible in all ABAP programs, local types are visible only inside the defining program.

When you define a local type with the `TYPES` statement, or a data object with the `DATA` statement, you can refer with the `TYPE` clause to all data types that are visible in the context of the definition. Or you can use the `LIKE` clause to refer to another data object that is visible within this context. In both cases, the newly defined data type or object inherits the attributes of the type or object that is being referred to.

For the elementary data types, ABAP provides predefined patterns that you can use to define your own types or objects. The predefined data types with fixed lengths are:

- Numeric types F (floating point), I (integer), and P (packed number in BCD format)
- Character types C (character), N (numeric text), D (date), and T (time)
- Hexadecimal type X (hexadecimal field)

The predefined data types with variable lengths are:

- Character string STRING
- Byte string XSTRING

While the length of data objects with fixed length types is constant at runtime, the length of data objects with variable length types will vary according to their contents.

Complex types are composed from any other data types that can be elementary, itself complex, or reference types. While structures are sequences of other data types, internal tables consist of a dynamic sequence of lines, all with the same data type. There are no predefined complex types — you must define them yourself in a program or in the ABAP Dictionary. For example, the ABAP statement for defining an internal table is:

```
TYPES|DATA itab TYPE|LIKE TABLE
OF line WITH key.
```

This statement creates a data type or data object `itab`, where the data type of the lines is taken from data type or object `line`. The `key` addition specifies the internal table's key.

Reference variables contain references (pointers) that can either point to data objects or to instances of classes in ABAP Objects. There are no predefined reference types, but you must define them yourself in a program or in the ABAP Dictionary. The ABAP statement for defining a reference is:

```
TYPES|DATA myref TYPE REF TO
DATA|otype.
```

This statement creates a reference type or reference variable `myref`. Pointers in variables of that type may either point to data objects or to instances of all classes that are specializations of the object type `otype`.

Going back to the code listing featured in Figure 6, we can now understand its data declaration. A data object named `wa_spfli` is declared by referring to a type `spfli`. Since there is no type of this sort defined locally in the program, `spfli` must be a type from the ABAP Dictionary. As can be seen in the `WRITE` statement, `wa_spfli` is a structure with different components, and `spfli` is a structured data type. As for all repository objects, you can navigate from the ABAP Editor to the type's definition by double-clicking its name.

Local Data and Global Data in ABAP Programs

When we speak about data in ABAP programs, it is very important to look at the places where you can declare data, and where that data is visible. The ABAP Workbench supports this with its forward navigation. If you double-click on names of types or data anywhere in an ABAP program, the Workbench leads you to the place of definition.

There are exactly three contexts where data objects and types can be defined inside ABAP programs:

- **Local data in procedures**

  When you use the statements `DATA` or `TYPES` in a procedure (subroutine, function module, or method), you define local data or types in that procedure. They are visible only inside the procedure and live as long as the procedure is executed. The parameters of the procedure's parameter interface also behave like local data.

- **Attributes in classes**

  When you use the statements `DATA` or `TYPES` between the statements `CLASS` and `ENDCLASS`,

you define attributes or types of that class. In classes, you must define explicitly the visibility of attributes and types. In general, you define private attributes that are visible only in the methods of the same class, but public attributes that are visible for the outside client are also possible. For class data we distinguish between instance and static data. While instance data is bound to the objects of a class, static data is always available.

- **Global program data**

  When you use the statements DATA or TYPES anywhere else in an ABAP program, especially if you use them in event blocks or dialog modules, you define global data or types of that program. They are visible in all the program's processing blocks that are defined behind their definition.

Working with global program data is dangerous, especially in large and complex programs. Since global data can be changed from everywhere, it is not easy to keep it consistent. Imagine a function pool with global data declarations in its declaration part. Any function module of that function pool can work with that data. A function pool is loaded into memory when the first of its function modules is called. Then it stays in memory and holds its global data as long as the calling program is running. If in the same program, maybe much later, another function module of the same function pool is called; this one will find the old contents of the preceding function modules in the global data. In some cases this effect might be desired, but it can also lead to errors, especially if you forget to initiate the data before using it.

This is why you should try to work with encapsulated local data in classes or procedures as much as possible. Except for a few cases, there is no real need to work with global data in ABAP. In cases where you must work with global data — for example, to transport data from and to a screen or to receive data from a logical database — for the sake of readability, be sure to define all global data in the declaration part at the top of the program, and never inside event blocks or dialog modules. A source code modularization with a TOP Include supports this issue.

Figure 10 shows how the program in Figure 6 can be rewritten in order to get rid of the global data. Now, the data object `wa_splfi` is declared locally in a subroutine `get_data`. Another subroutine, `output_data`, has a parameter interface and writes the data to the screen.

Screens

The preceding sections gave you an overview of the structure, execution, and data of ABAP programs embedded in the ABAP runtime environment at the application server. This section explains how ABAP programs communicate with the user via screens.

As a user of an R/3 system, you are always confronted with screens. From the moment you log on, you see a screen and you must perform actions on this screen. All those screens are components of ABAP programs.

Generally, you define the screens of an ABAP program with the Screen Painter tool of the ABAP Workbench. In the classical programming model, general screens were mainly used for dialog programming with module pools. The use of screens, however, is not restricted to module pools. Executable programs and function pools can contain and use screens, too. Besides the general method, there are also special screens that can be defined with ABAP statements within the program. These are the selection screens and lists, which are widely used in classical reporting.

General Screens

In the R/3 system, screens are program objects that consist of two parts. First, they have a layout that

Figure 10 Simple ABAP Program with No Global Data

```
REPORT select_and_write_with_forms.
*********************************************************************
* Processing blocks
*********************************************************************
START-OF-SELECTION.
  PERFORM get_data.
*********************************************************************
FORM get_data.
  DATA wa_spfli TYPE spfli.
  SELECT SINGLE * FROM spfli
                  INTO wa_spfli
                  WHERE carrid = 'UA' AND connid = '0941'.
  PERFORM output_data USING wa_spfli.
ENDFORM.
*********************************************************************
FORM output_data USING l_spfli TYPE spfli.
  WRITE: l_spfli-airpfrom, l_spfli-airpto, l_spfli-fltime.
ENDFORM.
*********************************************************************
```

defines the frontend appearance of the window that is presented to the user. Second, they have a flow logic that is executed on the backend by the application server. The screen flow logic is a program layer between the frontend and the actual ABAP application program at the backend. The language used to program screen flow logic has a similar syntax to ABAP, but is not part of ABAP itself. Unlike ABAP programs, the screen flow logic contains no explicit data declarations. You define the screen fields by placing elements on the screen mask instead. When you define screen fields by referring to data types in the ABAP Dictionary, the runtime environment automatically creates dialogs for field help, input help, and error handling that depend on the semantics of the data type in the dictionary.

The screen flow logic is similar to an ABAP program in that it contains processing blocks. These processing blocks are event blocks that are triggered by the ABAP runtime environment. The most important event blocks are:

- **`PROCESS BEFORE OUTPUT`**

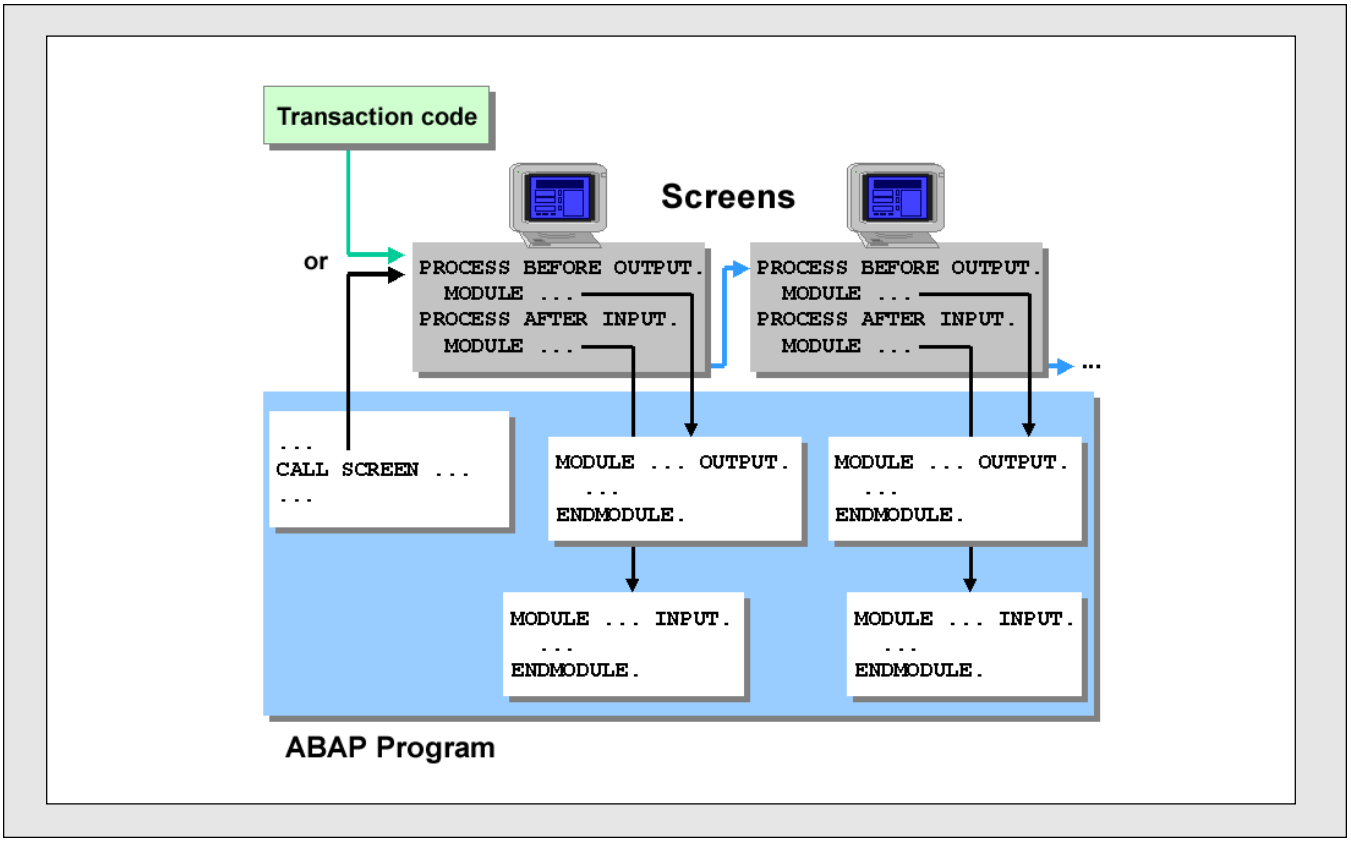
  The respective event (PBO) is triggered after the `PROCESS AFTER INPUT` (PAI) processing of the previous screen and before the current screen is displayed.

- **`PROCESS AFTER INPUT`**

  The respective event (PAI) is triggered when the user chooses a function on the current screen.

The main task of these processing blocks is to call ABAP dialog modules using the `MODULE` statement. During the PBO event, you can call any dialog module in the ABAP program that is marked with the addition `OUTPUT.` In the PAI event, you can call any dialog module program that is marked with the addition `INPUT`. The screens of an ABAP program can share the dialog modules of that program. You use the dialog modules called during PBO to prepare the screen and the dialog modules called during PAI to react to the user input.

Figure 11 Calling a Screen Sequence



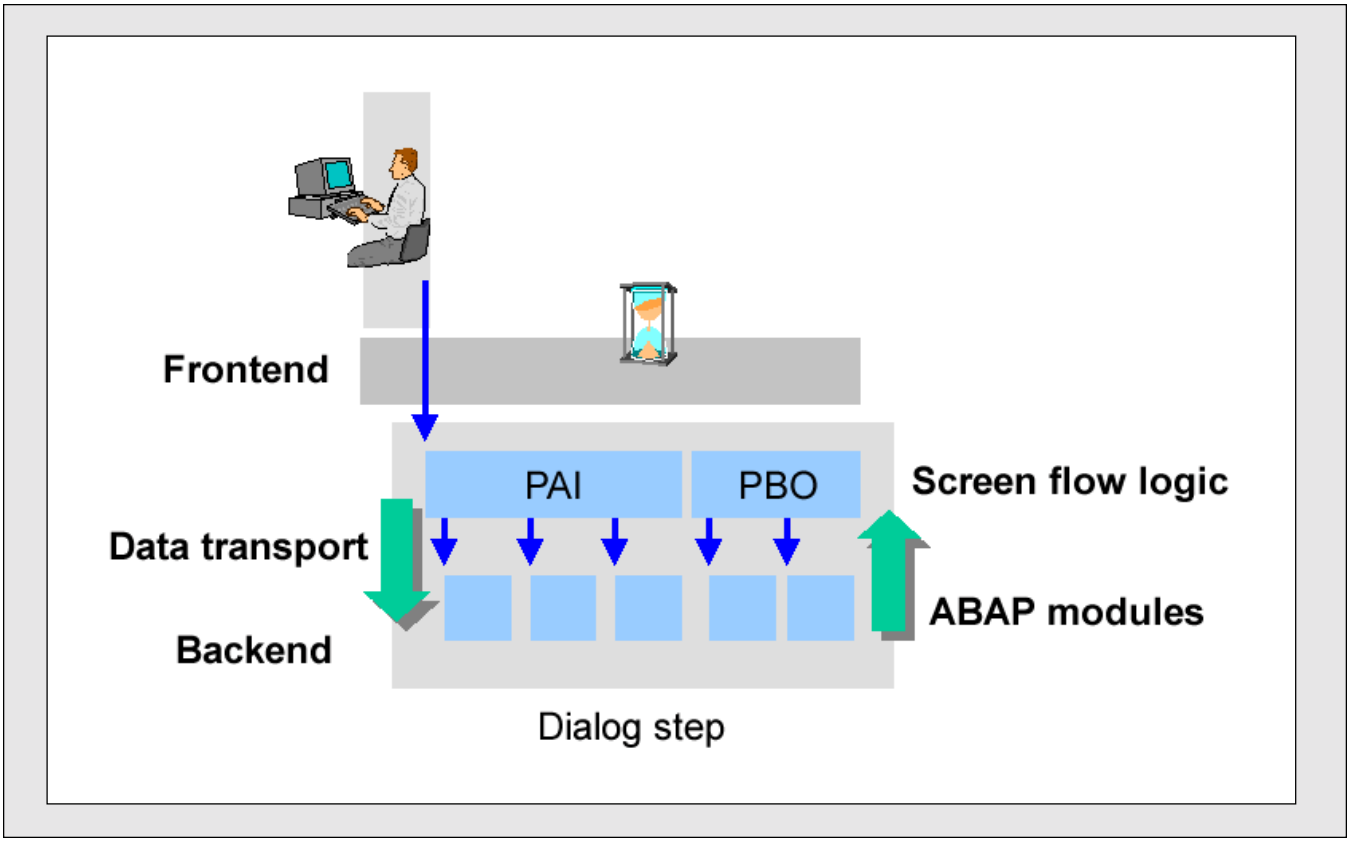
Each screen of an ABAP program has a unique screen number. The screens of an ABAP program can be combined to form screen sequences. Screen sequences are either built statically by setting the following screen in the Screen Painter or dynamically by overriding the static setting in the ABAP program. The last screen of a screen sequence is always the one where the following screen is set to zero. The smallest screen sequence is a single screen that is directly followed by screen zero. **Figure 11** shows how you call screens or, to be precise, a screen sequence.

You call a screen sequence either by using a transaction code from outside the ABAP program, or by using the `CALL SCREEN` statement in the same ABAP program. As you already know, a transaction code is always connected to a screen of an ABAP program. When you use a transaction code or the `CALL SCREEN` statement, the runtime system triggers the PBO event, and the corresponding coding is executed in the screen's flow logic (shown in gray). After processing PBO, the screen is displayed at the frontend where a user action triggers PAI. After processing PAI, the runtime system triggers PBO of the next screen in the sequence. This is repeated until the number of the next screen in the sequence is zero. Depending on how the first screen was called, the program either terminates, and control returns to the point from where the transaction code was used, or the program continues processing directly after the `CALL SCREEN` statement.

A user can interact in various ways with screens. We distinguish between actions that trigger PAI and those that don't. In general, filling input fields with values does not trigger PAI. Actions that do trigger PAI include:

- Choosing a pushbutton on the screen.

Figure 12 Data Transport Between the Screen and the ABAP Program



- Choosing a specially prepared check box or radio button on the screen.
- Choosing a function in the menu, standard toolbar, or application toolbar.
- Choosing a function key on the keyboard.

What all of these actions have in common is that they are all linked to a function code. The function codes of elements on the screen are defined with the Screen Painter. The function code in menus, standard toolbar, application toolbar, or those assigned to function keys are defined in a GUI status. A GUI status is an independent component of an ABAP program that provides the user with a range of functions in the user interface of the screen. It is defined with the Menu Painter tool of the ABAP Workbench. One GUI status can be reused by several screens. Each screen automatically contains a special field that is filled with the corresponding function code when the user chooses a function. As for all other screen fields, the contents of that field can be evaluated in the ABAP program in order to react to the user's input.

Figure 12 shows the data transport between the screen and the ABAP program. During PAI, the system automatically transports all screen fields to identically named global ABAP program fields. At the end of the last PBO module, and before the screen is displayed, all of the data is transported from the ABAP program to any identically named fields in the screen. By standard, all screen data is transported immediately before PAI processing starts. But for the dedicated handling of screen fields — for example, to carry out an error dialog — you can control the moment at which data is passed from screen fields to their corresponding ABAP fields by using the `FIELD` statement in the screen flow logic.

The time when the ABAP dialog modules process the data for the screens is called a "dialog step." A dialog step extends between a user action on one screen and the appearance of the following screen. This time comprises the processing of PAI and PBO of two successive screens. Note that during a user dialog with the R/3 system, only the dialog steps occupy the application server, and not the time where the screen accepts a user's input.

Special Screens

ABAP provides two types of special screens: *selection screens* and *lists*. These screens and their flow logic are generated from ABAP statements. In this case, you do not have to work with the Screen Painter.

Selection screens are special screens used to enter values in ABAP programs. You can define selection screens in the global declaration part of any ABAP program that can contain screens using the following special declaration statements: `SELECTION-SCREEN` defines and formats selection screens, and `PARAMETERS` and `SELECT-OPTIONS` define input fields for single values or intervals. An executable program or a logical database may have a standard selection screen with the reserved number 1,000. You call a selection screen in the same ABAP program using the `CALL SELECTION-SCREEN` statement. If the program is an executable, the ABAP runtime environment automatically calls the standard selection screen after the `INITIALIZATION` event. The PBO and PAI handling of selection screens is encapsulated in the ABAP runtime environment and user actions on selection screens trigger the `AT SELECTION-SCREEN` events.

Lists are output-oriented screens that display formatted, structured data. Data in list format can be sent to the R/3 spool system for printing. You define lists using a special set of statements (`WRITE`, `SKIP`, `ULINE`, and so on). With these statements, a list is created and buffered on the application server. An internal system program called the "list processor" is responsible for displaying a list and for interpreting user actions on the list. You call the list processor with the `LEAVE TO LIST-PROCESSING` statement. If the program is an executable, the ABAP runtime environment automatically calls the list processor after triggering the last event of the program. As for selection screens, the PBO and PAI handling is encapsulated in the runtime environment and user actions on a list trigger events (e.g., `AT LINE-SELECTION`). Output statements in the respective event blocks create detail lists, which are then automatically displayed at the end of the event block. You can create up to 19 detail lists for a single basic list. The user can navigate between the different levels by choosing functions from the list's GUI Status.

Using Screens

From the facts you learned in this and the preceding sections, it should be clear that the usage of the different types of screens is not restricted to special program types. General screens are not restricted to module pools and selection screens, and lists are not restricted to executable programs. Which screen you use depends merely on which dialog your program shall perform with the user. So you might start a module pool with a transaction code connected to a general screen, send a selection screen during the PBO processing of that screen, and define and process a list during PAI. Vice versa, you might start an executable program, take advantage of its standard selection screen, and after processing it, call a sequence of general screens. And finally, don't forget that any kind of screens can also be defined in function pools. For example, by defining selection screens in function pools, you can completely encapsulate data selections, including the respective user dialogs.

The program in **Figure 13** adds a selection screen to the program we just reviewed. This screen is defined as screen number 500 in the global data declaration part. It has two input fields —

Figure 13 Adding a Selection Screen to Our Simple ABAP Program

```
REPORT select_with_selection_screen.
************************************************************************
* Selection Screen
************************************************************************
SELECTION-SCREEN BEGIN OF SCREEN 500 AS WINDOW.
PARAMETERS: p_carrid TYPE spfli-carrid VALUE CHECK,
            p_connid TYPE spfli-connid VALUE CHECK.
SELECTION-SCREEN END OF SCREEN 500.
************************************************************************
* Processing blocks
************************************************************************
START-OF-SELECTION.
  PERFORM get_data.
************************************************************************
FORM get_data.
  DATA wa_spfli TYPE spfli.
  CALL SELECTION-SCREEN 500 STARTING AT 10 10.
  SELECT SINGLE * FROM spfli
                  INTO wa_spfli
                  WHERE carrid = p_carrid AND connid = p_connid.
  PERFORM output_data USING wa_spfli.
ENDFORM.
************************************************************************
FORM output_data USING l_spfli TYPE spfli.
  WRITE: l_spfli-airpfrom, l_spfli-airpto, l_spfli-fltime.
ENDFORM.
************************************************************************
```

`p_carrid` and `p_connid` — and is formatted as a dialog window. The type of the input fields refers to types in the ABAP Dictionary. The runtime system provides help and error dialogs for these fields. The screen is called in the subroutine `get_data`. The user can enter single values that are then used in the `WHERE` clause of the data selection.

ABAP Database Access

Looking back at Figure 4, we see that one topic is still missing in our overview — how ABAP programs access databases. This is what we will discuss now.

In the R/3 system, long-life data is stored in the tables of one relational database. Each database system has a programming interface that allows you to access the database tables using SQL (Structured Query Language) statements. The ABAP runtime environment provides a database interface to make the R/3 system independent of the database system and differences in the SQL syntax between various databases. ABAP programs communicate with the database by means of this interface. The database interface converts all of the database requests from the R/3 system into the correct Standard SQL statements for the database system. There are two ways of accessing the database from an ABAP program — with Open SQL or Native SQL statements.

Open SQL

Open SQL statements, a subset of Standard SQL, are fully integrated in ABAP. They allow you to access data irrespective of the database system that the R/3 installation is using. Open SQL consists of the Data Manipulation Language (DML) part of Standard SQL; in other words, it allows you to read (`SELECT`) and change (`INSERT`, `UPDATE`, `DELETE`) data.

Open SQL goes beyond Standard SQL to provide statements that, in conjunction with other ABAP constructions, can simplify or speed up database access. It also allows you to buffer certain tables on the application server, saving excessive database access.

Native SQL

Native SQL is only loosely integrated into ABAP. It allows access to all the functions contained in the programming interface of the respective database system, but unlike Open SQL statements, are not checked and converted. They are sent directly to the database system. When you use Native SQL, the function of the database-dependent layer is minimal. Programs that use Native SQL are specific to the database system for which they were written. When writing R/3 applications, you should avoid using Native SQL wherever possible. It is used, however, in some parts of the R/3 Basis system — for example, for creating or changing database table definitions in the ABAP Dictionary.

The Role of the ABAP Dictionary

Open SQL contains no statements from the DDL (Data Definition Language) part of Standard SQL, because normal application programs should not create or change their own database tables. Instead, the ABAP Dictionary tool of the ABAP Workbench uses DDL internally, allowing you to create and administer database tables. Open SQL statements can only access tables that are created by the ABAP Dictionary tool. For each database table, the tool simultaneously creates an equally named data type in the ABAP Dictionary. This type is structured and its components have the same types and names as the database table's columns. You can use these types for creating data objects as work areas for Open SQL statements in your ABAP programs.

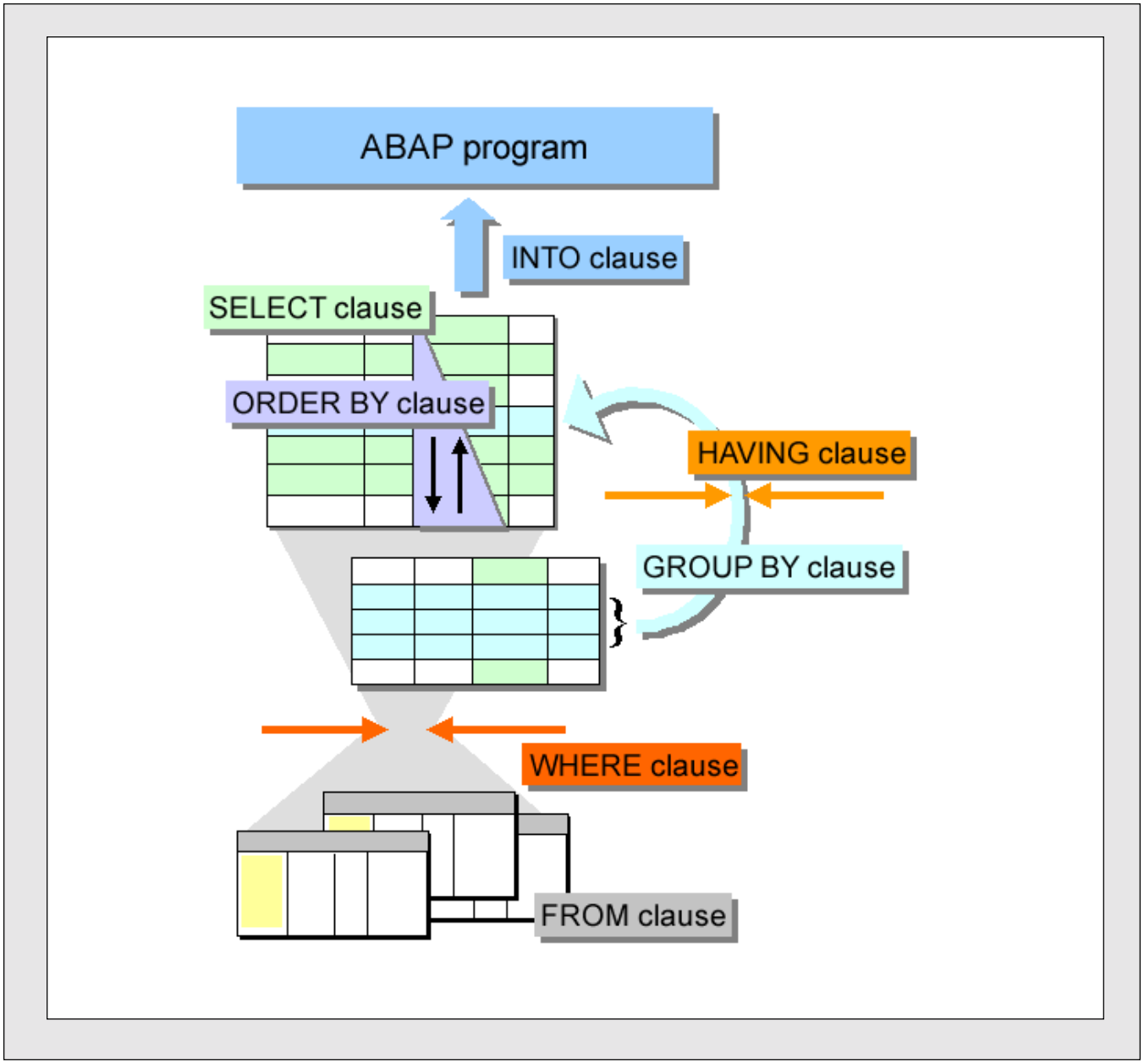
You should never confuse the roles of types, data objects (work areas), and database tables. Note that in our example programs, the data object `wa_spfli` is declared by referring to the type `spfli` in the dictionary while the Open SQL statement `SELECT` accesses the equally named database table `spfli` and reads data into the work area `wa_spfli`. Needless to say, assigning the work area the same name as the database table can be a source of confusion. In R/3 releases prior to 4.0, it was necessary to declare such equally named work areas with the `TABLES` statement in order to use Open SQL. Now you can do so without the `TABLES` statement and this statement is obsolete for database accesses.

The SELECT Statement

Take a look now at **Figure 14**. In this example of ABAP's Open SQL, the `SELECT` statement is divided into a series of simple clauses, each of which has a different part to play in selecting, placing, and arranging the data from the database.

Open SQL contains no statements from the DDL (Data Definition Language) part of Standard SQL, because normal application programs should not create or change their own database tables. Instead, the ABAP Dictionary tool of the ABAP Workbench uses DDL internally, allowing you to create and administer database tables. Open SQL statements can only access tables that are created by the ABAP Dictionary tool.

Figure 14 Functions of the SELECT Statement



The SELECT clause defines the structure of the data you want to read — i.e., defining whether you want to read one line or several, which columns you want to read, and whether identical entries are acceptable or not. The INTO clause determines the ABAP fields into which the selected data is to be read. The FROM clause specifies the database table or tables from which the data is to be selected.

The WHERE clause specifies which lines are to be read by specifying conditions for the selection. The GROUP BY clause produces a single line of results from groups of several lines. The HAVING clause sets logical conditions for the lines combined using GROUP BY. The ORDER BY clause defines a sequence for the lines resulting from the selection.

Figure 15 A SELECT Statement Example

```
SELECT    p~carrid p~connid f~fldate b~bookid
INTO      CORRESPONDING FIELDS OF TABLE itab
FROM      ( ( spfli AS p
              INNER JOIN sflight AS f ON p~carrid = f~carrid AND
                                         p~connid = f~connid      )
              INNER JOIN sbook   AS b ON b~carrid = f~carrid AND
                                         b~connid = f~connid AND
                                         b~fldate = f~fldate      )
WHERE    p~cityfrom = 'FRANKFURT' AND
         p~cityto   = 'NEW YORK'  AND
         f~seatsmax > f~seatsocc
ORDER BY p~carrid p~connid.
```

Figure 15 shows a SELECT statement and the usage of some of its clauses. Note that this code listing consists of a single ABAP statement. The FROM clause assigns alias names to the database tables spfli, sflight, and sbook, and links the columns carrid, connid, fldate, and bookid of these tables as an inner join. With the INTO clause, a list of booking numbers for all flights from Frankfurt to New York that are not fully booked are written into an internal ABAP table itab.

When you program database accesses with the SELECT statement, you should always keep an eye on performance. The performance of an ABAP program is largely determined by the efficiency of its database accesses. It is therefore worth analyzing your SQL statements closely. To help you do this, you can use the *Performance Trace* and *Runtime Analysis* tools of the ABAP Workbench.

ABAP Objects

In the final section of this article, we'll take a brief look at ABAP Objects and how this object-oriented concept fits in with the classical concepts we have been discussing.

ABAP Objects is a nearly 100 percent upward-compatible extension to the ABAP language that includes a full set of object-oriented (OO) features. The extension consists of a set of new statements that define classes or interfaces, and which handle objects. These statements can be used in any kind of ABAP program.

The basis for ABAP Objects is classes. Classes are coding sequences that define the components of objects. Possible components are attributes (data), methods (functions), and events. Interfaces allow you to decouple the declaration of public components from the class definition. Classes and interfaces can either be defined locally in an application program, or globally in the class library using the Class Builder tool of the ABAP Workbench. Similar to the function library, the class library is part of the R/3 repository.

Objects are instances of classes and are accessed only via reference variables. Object reference variables contain references (pointers) to objects and must be declared before creating an object. There is almost no difference between creating objects based on global classes and creating them on local classes. Instances (objects) can be created with the CREATE OBJECT statement only from classes, not from interfaces. Nevertheless, object reference variables can refer either to classes or to interfaces.

The important concept of polymorphism, meaning that you can point with one type of reference variable to objects of different classes, is provided by the concepts of single inheritance and interfaces. Class reference variables that are typed with reference to a superclass can point to objects of all subclasses, and interface reference variables can point to objects of all classes that implement the respective interface.

ABAP Objects fits smoothly into the concepts of ABAP as discussed in this article. As you have already seen in the preceding section, classes are handled syntactically as processing blocks, and methods are procedures that are very similar to function modules. ABAP programs with classes and methods are embedded in the ABAP runtime system just as any other ABAP program.

You can instantiate and handle objects during the execution of all classical ABAP programs. But with ABAP Objects, there will also be another kind of program execution that decouples object-oriented programming from the runtime event-oriented programming to a great extent. From Release 4.6C on, you will be able to create transaction codes that are directly linked to methods of either global or local classes. When you call such a transaction, the system loads the program where the class is defined into the memory, automatically creates an instance of that class, and then executes the method.

The procedural statements that you use to code the functionality of methods are the same ones you use to code any other processing block. Nevertheless, the syntax is stricter in some places inside ABAP Objects in order to purge some obsolete features from the language. For example, in classes you cannot use header lines for internal tables, or the `TABLES` statement.

If you have some experience in ABAP programming, you can learn the concept and syntax of ABAP Objects in a few hours. With that knowledge you are able to use methods that are provided in public classes as you have used function modules before. For example, since Release 4.6, the ability to display GUI controls for pictures, trees, text editors, etc., on screens is encapsulated in the public classes of the so-called Control Frame Work (CFW). By using these classes in existing ABAP programs, you can strongly improve the layout and usability of your applications.

The program in **Figure 16** gives you a sense for the syntax that characterizes ABAP Objects. The function of the program is the same as the function of the program shown back in Figure 10. Here, however, we use ABAP Objects within the framework of an executable program. After starting the program, a static method `start` of a class `main` is called during the `START-OF-SELECTION` event. There, an object of class `flight` is created and its methods are called. You can type the program in the ABAP Editor and execute it in R/3 Release 4.5 or higher. ABAP Objects is fully integrated in the ABAP Debugger, and you might debug the program to learn more. You may want to try replacing the

Figure 16 Sample Program with Classes and Objects

```
REPORT select_and_write_with_objects.
*---------------------------------------------------------------------*
* Processing blocks
*---------------------------------------------------------------------*
CLASS flight DEFINITION.
  PUBLIC SECTION.
    METHODS: get,
             output IMPORTING l_spfli TYPE spfli.
  PRIVATE SECTION.
```

(continued on next page)

Figure 16 (continued)

```
    DATA wa_spfli TYPE spfli.
ENDCLASS.
CLASS flight IMPLEMENTATION.
  METHOD get.
    SELECT SINGLE * FROM spfli
                   INTO wa_spfli
                   WHERE carrid = 'UA' AND connid = '0941'.
    CALL METHOD me->output EXPORTING l_spfli = wa_spfli.
  ENDMETHOD.
  METHOD output.
    WRITE: l_spfli-airpfrom, l_spfli-airpto, l_spfli-fltime.
  ENDMETHOD.
ENDCLASS.
*********************************************************************
CLASS main DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS start.
ENDCLASS.
CLASS main IMPLEMENTATION.
  METHOD start.
    DATA flight TYPE REF TO flight.
    CREATE OBJECT flight.
    CALL METHOD flight->get.
  ENDMETHOD.
ENDCLASS.
*********************************************************************
START-OF-SELECTION.
  CALL METHOD main=>start.
*********************************************************************
```

definition of the classical list with the `WRITE` statement by using methods of the global class `CL_GUI_ALV_GRID` instead. This class is an example of classes in the Control Frame Work. It serves as an **A**BAP **L**ist **V**iewer. In future releases, we plan to implement many standard components (e.g., logical databases in such classes).

With the new SAP initiatives EnjoySAP and mySAP.com, you can take full advantage of ABAP Objects by programming or using standardized interfaces for the new presentation layers that will be supported then.

Let me end this brief ABAP Objects discussion with a note of caution. Knowing how to use an object-oriented language is *not* the same as knowing how to develop a complete object-oriented application. Object-oriented development has logical advantages that are independent of the concrete implementation. The most important (and most time-consuming) part of developing a real object-oriented application is the object-oriented modeling. Object-oriented languages such as ABAP Objects simply provide the coding support for the implementation of an object-oriented model.

Conclusion

Following a simple example, we took you on a tour from the ABAP runtime environment, over program structure and program execution, to data types and objects. You learned the principles of screen handling and database accesses. Finally, we gave you a short introduction to ABAP Objects, the basis of the new ABAP programming model, which in coming releases will have at least the same importance as the classical programming model.

Much of ABAP's strength comes from the fact that you can program small applications in a complex client/server environment with a few simple statements. The background information in this article will be helpful for you if you want to program more complex applications or try to understand existing programs.

Let us pick up the example of local/global data another time. When you find data or type declarations in an event block or a module pool, you know now that these declarations are not local, but global. Avoid such mistakes when you program yourself. Use procedures to create encapsulated local data. For this purpose, we recommend using local classes and their methods instead of subroutines. Even when implementing only internal modularizations, you will profit from a cleaner and safer programming style that is enforced by the improved syntax rules and type checks of ABAP Objects.

And last, but not least, programming with ABAP Objects is just more fun!

Horst Keller studied and received his doctorate in physics at the Darmstadt University of Technology, Germany. After doing research in several international laboratories, he joined SAP in 1995, where he worked in the ABAP Workbench Support Group for three years. During that time, he authored the books on ABAP programming in the printed ABAP Workbench Documentation Set for R/3 Release 3.0. With R/3 Release 4.0, he started to reorganize the ABAP online documentation in order to realize an integrated view of the programming environment.

Horst is now a member of the ABAP language group, where he is responsible for information rollout. He is documenting the ABAP language with an emphasis on ABAP Objects. He also develops and teaches classes on ABAP programming, and is writing productive programs that display the ABAP documentation and the corresponding examples in the R/3 Basis system. Horst can be reached at horst.keller@sap.com.