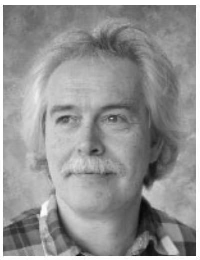
An insider's guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs

Part 2 — best practices

by Andreas Blumenthal and Horst Keller



Andreas Blumenthal
Vice President,
SAP NetWeaver
Application Server ABAP,
SAP AG

Horst Keller
Knowledge Architect,
SAP NetWeaver
Application Server ABAP,
SAP AG

(Full bios appear on page 28.)

Dedication

This article is dedicated to the memory of Axel Kurka, who for many years pushed ahead SAP technology, and especially ABAP technology, in his roles as Developer, Development Manager, Product Manager, and last but not least as "Dr. ABAP" in the *SAP Technical Journal*. Axel died much too early in August 2005, but his actions have influenced SAP technology forever.

As application needs have grown and changed, so have programming languages to meet those needs, and ABAP is no different. Over the years ABAP has evolved along with developer needs to offer new and more efficient ways to accomplish programming tasks. As ABAP has evolved, the combination of new features and those preserved for downward compatibility has led to an exponential increase in the options available to developers. For those who are new to ABAP, the sheer number of possibilities can be overwhelming. And for those who have been using ABAP for years, the temptation to stick with old, less efficient ways of doing things is hard to resist when that is what you are familiar with (and you know that it will work).

This is the second article in a three-part article series that presents to both ABAP newcomers and old-timers basic guidelines for the effective use of the many possibilities ABAP provides to application developers to create well-structured, high-performing, robust business applications.

In the first part of this article series,[1] we laid some groundwork by first introducing fundamental rules and formal criteria for modern ABAP programming. The most important of these were:

[1] "An insider's guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs: Part 1 — fundamental rules and formal criteria" (*SAP Professional Journal*, January/February 2006).

- Use ABAP Objects
- Use Unicode-enabling

In this second part, we'll drill down into some ABAP programming nitty-gritty and present suggestions for how best to use the language elements allowed in Unicode-enabled ABAP Objects. In particular, we will examine best practices for:

- ABAP Objects, modularization, and program flow
- Declarations
- Processing data

> ***Note!***
>
> This article, like the first, applies to SAP Web Application Server (SAP Web AS) 6.10 and higher. Although some of the concepts mentioned in this article are available only as of SAP NetWeaver Application Server (SAP NW AS) ABAP 7.0 in SAP NetWeaver 2004s (the successor to SAP Web AS ABAP 6.40 in SAP NetWeaver '04), many of the same principles apply and you should be able to follow along easily.

> ***Note!***
>
> Many of the guidelines presented in this article assume that you are following our first and foremost recommendation, presented in Part 1 of this article series — to use the ABAP Objects programming model, which is easier to use and provides enhanced programming capabilities, wherever possible.[2]

[2] For more on why ABAP Objects is the recommended approach, see Part 1 of this article series in the January/February 2006 issue of *SAP Professional Journal* and also the article "Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Programmer Should Give It a Second Look" in the September/October 2004 issue of *SAP Professional Journal*.

As you will see, there are often various ways to accomplish a given task, and there are a number of language elements that, while not forbidden syntactically in Unicode-enabled classes, should no longer be used. This article serves as a list of "tips and tricks" that will help you choose the most appropriate variants for the task at hand, and as a list of language elements that (while still valid) should be regarded as deprecated.

Best practices for ABAP Objects, modularization, and program flow

The following sections provide recommendations for:

- Using ABAP Objects
- Modularizing your programs
- Declaring the visibility of instance constructors
- Defining the parameter interface of a procedure
- Defining control structures for processing blocks

Using ABAP Objects

As stated in Part 1 of this article series, ABAP Objects is the recommended programming model. But how do we define the term *programming model*? As in other object-oriented (OO) programming languages, the use of methods and classes ranges from real model-driven usage (UML-based programming) to simple procedural method calling (simply using methods of classes instead of function modules or subroutines).

Recommendation

If you are not an "OO wizard," there is no reason to become a slave to the academic OO paradigm. Instead, use OO with common sense, appropriate to your aims. Especially, you can and should exploit all useful and proven 4GL-features of ABAP (internal tables and Open SQL, for example) in your method implementations.

If you do not intend to fully exploit an OO design phase, a practical approach to OO programming can be a conservative approach, where the watchword is encapsulation (defensive programming). Start always by being as restrictive as possible and ease the restrictions only if really needed:

- Declare classes as `FINAL`.
- Restrict the number of public components. All components that *can* be private *should* be declared `PRIVATE`.
- If you use public attributes, they should be declared `READ-ONLY`.
- Consider using private instantiation for your classes (`CREATE PRIVATE`) and offer appropriate factory methods.[3] This allows you to:
  - Control the instantiation of your class from within the class.
  - Control the parameters passed to the constructor.
  - Handle the instances of your class within the class.
  - Share the instances of your class effectively by passing the same object reference to different clients when appropriate.
  - Release clients from activating and deactivating event handlers.

Other recommendations for the practical use of ABAP Objects that give your programs an adequate structure include the following:

- For purposes of internal modularization, exploit local classes in class pools. This is preferable to private methods of the global class. With that, you can keep your main classes slim and their interfaces readable.
- For internal modularization of a class pool, prefer instance methods in local singletons[4] to static methods of local classes. Two important reasons to prefer instantiation over the direct use of classes via static methods are that you can:
  - Use inheritance and method redefinitions,[5] if needed.
  - Control the moment of the instantiation and the lifetime of the object.
- Use inheritance moderately. When in doubt, prefer interfaces to achieve polymorphism. Inheritance is one of the most powerful but also one of the more subtle features object orientation offers. If you use inheritance (for reusing method implementations, for example), keep the paths of your inheritance tree short and terminate them with final classes. For the reuse of interfaces, exploit the concept of interface composition in favor of inheritance.
- When you are addressing objects of classes that implement standalone interfaces, you should generally do so by using interface reference variables for accessing the interface components (using `iref->comp`). If the class provides an alias for an interface component, this component counts as a class component and you might address it via a class reference variable and the alias name. But please note that the interface component selector (~) should never be used by the outside user to address an interface component of an object via a class reference variable (don't use `cref->intf~comp`). The reason is that the outside user's view of a class should be defined by the public interface offered by the class. The public interface of a class consists of its own class components and can be enhanced by interfaces. The enhancement of the public interface of a class with interface components does not introduce a new hierarchy level. All components of the class lie on the same level but are offered from different

[3] A factory method is a method — normally static — of a class that creates an object of that class and returns a reference that points to that object.

[4] A singleton is an object-oriented pattern. A class defined according to the singleton pattern is implemented in such a way that exactly one object can be instantiated from that class. A possible way to do so is to define a class with private instantiation and to use the `CREATE OBJECT` statement in the static constructor only.

[5] Remember that you cannot redefine static methods in ABAP Objects.

Accessing interface components

This example shows how the interface components of objects (i.e., the components of objects that are components of some interface implemented by the object's class) can be accessed, and which access is recommended. The interface method `imeth` of the object referenced by `cref1` should be accessed via an interface reference variable only. In the object referenced by `cref2`, it can be accessed alternatively via its alias name `meth`. In all cases, the use of the interface component selector ~ is not recommended, because it introduces a hierarchical view on components that are part of the same level.

```
INTERFACE intf.
  METHODS imeth.
ENDINTERFACE.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES intf.
ENDCLASS.

CLASS c1 IMPLEMENTATION.
  METHOD intf~imeth.
    ...
  ENDMETHOD.
ENDCLASS.

CLASS c2 DEFINITION.
```

parts of the public interface. This is expressed by using the two-level expression `iref->comp` instead of the three-level `cref->intf~comp`, where `iref` and `cref` denote which part of the public interface is addressed. Therefore, `iref->comp` is more than just a shortcut for `cref->intf~comp`. Accessing a class via an interface reference variable always expresses that you are interested in the aspect of the class that is offered by the interface. (See the sidebar "Accessing interface components" above.)

- Don't forget the possibility of using events instead of direct method calls. If you must decouple caller from callee, there is no need to implement mechanisms — e.g., handler tables — for that on your own.
- Keep an eye on the memory consumption of objects. Release unneeded objects to the garbage collector — the garbage collector can only delete an object if the object is no longer referenced. Be aware that references to objects can exist in various contexts — e.g., in local fields of methods, in instance attributes of an object, in static attributes of classes, in global data objects of the program, and last but not least in event handler tables or frameworks. In the latter case, be aware that eventually you must use special methods of the respective framework to free objects.[6]

[6] In this context, it's worthwhile to call attention to system class `CL_ABAP_WEAK_REFERENCE`. Objects of that class represent *weak references* to an object. Unlike normal object references, a weak reference does not prevent the referenced object from being deleted when the garbage collector is executed. You might use this class if you need to create applications that simply monitor certain kinds of objects without keeping the monitored objects alive. An example of the need for such applications is the internal administration and monitoring of ABAP's shared objects in the shared objects memory.


```
  PUBLIC SECTION.
    INTERFACES intf.
    ALIASES meth FOR intf~imeth.
ENDCLASS.

CLASS c2 IMPLEMENTATION.
  METHOD intf~imeth.
    ...
  ENDMETHOD.
ENDCLASS.

DATA: iref  TYPE REF TO intf,
      cref1 TYPE REF TO c1,
      cref2 TYPE REF TO c2.

...

CREATE OBJECT: cref1,
               cref2.

iref = cref1.

iref->imeth( ).        "recommended
cref1->intf~imeth( ).  "not recommended

iref = cref2.

iref->imeth( ).        "recommended
cref2->meth( ).        "recommended
cref2->intf~imeth( ).  "not recommended
```

Modularizing your programs

If ABAP Objects becomes your recommended programming model, methods will naturally become the principal instrument for modularizing your programs. Although the following recommendations regarding method implementation have a more general rather than ABAP-specific character, we'll nevertheless not refrain from presenting these as ABAP coding guidelines.

Recommendation

Every method must be a coherent unit of work. Some very general guidelines for implementing methods are:

- Short is better than long.
- Modularize — don't atomize.
- Flat is better than deep.
- Avoid potential side effects.

From these, more practical recommendations can be derived:

- One-line methods should be an exception. As a rule of thumb, the number of lines should normally not be less than 5, the maximum number of lines should not exceed 50, and the number of declarative statements should not exceed 10. As a broader rule of thumb, a method should fit on a single printed page.
- The cyclomatic number of a method (number of branches due to tests) should not be greater than 10.[7]
- Be careful when you modify data that is global for a method inside the method. Possible non-local data that can be accessed directly for writing in a method (where "directly" means not via a reference) is:
    - All attributes of the class itself, including the protected attributes of super classes. A method can even write to private instance attributes of other objects of the same class.[8]
    - Public attributes of other classes that are not declared as `READ-ONLY`.
    - All global data of the ABAP program. As you will see in the upcoming guidelines for global data objects, the declaration of global data should be avoided under *any* circumstances. Nevertheless, there are exceptions to this rule, such as the interface work areas for classic Dynpros.

    Since operational coding is allowed in methods only, you must modify data that is global to methods within methods. In order to keep your methods free of side effects, avoid the careless modification of such global data in too many methods. Use only dedicated methods that can be recognized, for example, via their naming (`set_` methods). This is especially important for data that does not belong to the class itself. For example, the interface work area for a Dynpro should be set in one dedicated method of one dedicated class and only before calling the Dynpro.
- When you access data that is global for a method inside the method, it is a good idea to distinguish it from the method's local data via the appropriate selectors:
    - For attributes of the class itself, including the protected attributes of super classes, you can use the self reference `me` and the object component selector (`->`) for instance attributes, or the class name and the class component selector (`=>`) for static attributes.
    - Public attributes of other classes must be addressed via reference variables and the object component selector (`->`), or the class name and the class component selector (`=>`).
    - Global data of the program cannot be addressed with special selectors. Therefore it can be easily confused with attributes of the class itself or with local data of the method. This is an additional reason not to access the data in too many methods (see the sidebar "Addressing global data objects within a method" on the next page).

> ***Note!***
>
> Although we have not suggested strict naming rules (see the guidelines for coding style in Part 1 of this article series), some naming conventions can also be helpful here.

From the foregoing, a general rule emerges: you should restrict the scope of a method to a specific task. Refrain from working with too many attributes of the class or even global data inside a single

[7] When computing this number, we use a very simplistic approach — we don't count the `WHEN` conditions of a `CASE` structure or the `ELSEIF` conditions of an `IF` structure as individual tests, but each `CASE` or `IF` structure is counted as a single test. We are aware that the number of `WHEN` or `ELSEIF` conditions also increases method complexity and that a cyclomatic number of 10 without consideration of these conditions is a very rough criterion only.

[8] The level of encapsulation in ABAP Objects is the class, not the object.

Addressing global data objects within a method

The example below shows the addressing of data objects within a method in a chained `WRITE` statement.* The identifiers `a1` and `a2` address the respective local data objects of method `main`. `demo=>a1`, `me->a2`, and `a3` address the attributes of the class. `a4` addresses a global data object of the program. The global data objects `a1`, `a2`, and `a3` cannot be accessed from inside the method.

Now imagine that you see only the method because it is inside an INCLUDE program (remember the guidelines for source code organization in Part 1 of this article series), or because you have a printout. Simply looking at the method, you couldn't tell whether `a3` and `a4` are attributes of the class or global data. In the ABAP Workbench, you can double-click on the name to determine the place of declaration, but if you have only a printout, you are lost. Therefore, `a3` should be written as `me->a3` too. Also, a naming convention for global data (e.g., `g_a1`, `g_a2`) can be helpful here.

```
PROGRAM ...

DATA a1 TYPE string VALUE 'a1 global'.
DATA a2 TYPE string VALUE 'a2 global'.
DATA a3 TYPE string VALUE 'a3 global'.
DATA a4 TYPE string VALUE 'a4 global'.

CLASS demo DEFINITION.
  PUBLIC SECTION.
    METHODS main.
    CLASS-DATA a1 TYPE string VALUE 'a1 class'.
    DATA       a2 TYPE string VALUE 'a2 class'.
    DATA       a3 TYPE string VALUE 'a3 class'.
ENDCLASS.

CLASS demo IMPLEMENTATION.
  METHOD main.
    DATA a1 TYPE string VALUE 'a1 local'.
    DATA a2 TYPE string VALUE 'a2 local'.
    WRITE: / a1,
           / a2,
           / demo=>a1,
           / me->a2,
           / a3,
           / a4.
  ENDMETHOD.
ENDCLASS.
```

* Note that we use `WRITE` for list output here, in seeming contradiction to our guidelines for GUI programming in Part 1 of this article series. However, for simple tests and non-productive coding, `WRITE` can still play the same role in ABAP as `System.out.println(...)` does in Java, for example.

method, otherwise the method's implementation can quickly become confusing.

> ***Note!***
>
> Never call a procedure without making provision for appropriate exception handling (more on this in the guidelines for error handling in Part 3 of this article series).

Declaring the visibility of instance constructors

You know that the visibility of all components of a class must be defined by declaring each component in a visibility area (`PUBLIC SECTION`, `PROTECTED SECTION`, `PRIVATE SECTION`).[9] The same *seems* to hold for the declaration of the instance constructor that is declared with `METHODS constructor`. Unfortunately, this is not true. The true visibility of an instance constructor is declared with the `CREATE` addition of `CLASS DEFINITION`. By writing `PUBLIC`, `PROTECTED`, or `PRIVATE` behind `CREATE` in the `CLASS` statement, you denote who can create objects of that class using `CREATE OBJECT` and, therefore, who can invoke the instance constructor.

The declaration of the instance constructor with `METHODS constructor` within a visibility section is for syntactical reasons only (it must be declared somewhere). Up until SAP NetWeaver '04, the instance constructor could be declared in the public visibility section only. As of SAP NetWeaver 2004s, the instance constructor can be declared in all visibility sections that are the same or more general than the visibility for instantiation defined via the `CREATE` addition of `CLASS DEFINITION`. But remember, the true visibility is always defined by the `CREATE` addition.

Recommendation

Always declare the instance constructor in the visibility section that matches its true visibility. For a class declared with `CREATE PRIVATE`, the statement `METHODS constructor` should be placed in the `PRIVATE SECTION` (see the sidebar "Declaring the instance constructor" on the next page), for `CREATE PROTECTED` in the `PROTECTED SECTION`, and for `CREATE PUBLIC` in the `PUBLIC SECTION`. Then:

- The visibility section of the instance constructor protocols its technical visibility.
- You can refer to declarations of the same section in the interface of the constructor.

Defining the parameter interface of a procedure

When defining the parameter interface of a method (or a function module, when needed), you must determine:

- The **number** of formal parameters
- A formal parameter's **kind**[10]
  - `IMPORTING` — An *input parameter* to which the value of the actual parameter is passed at the time of a call. For input parameters that are passed by reference, no write access is allowed in the procedure. For input parameters that are passed by value, write access is allowed in the procedure.
  - `EXPORTING` — An *output parameter* whose value is available in the actual parameter after the procedure is completed. For output parameters that are passed by reference, the

[9] The visibility area defines the visibility of the components of a class and thus also the interfaces between the class and its users. For a more detailed discussion of this concept, see *The Official ABAP Reference*, by Horst Keller (SAP PRESS, 2005).

[10] A fifth kind of formal parameter, `TABLES`, can be used for passing internal tables only, but is obsolete. These parameters are syntactically forbidden for methods. Although still allowed for function modules, they should no longer be used.

Declaring the instance constructor

The example below shows the recommended declaration of the instance constructor in a class that is declared with `CREATE PRIVATE` as of SAP NetWeaver 2004s. Note that a declaration in the `PROTECTED` or `PUBLIC` section is also allowed (up until SAP NetWeaver '04, only the declaration in the `PUBLIC` section was allowed), but in that case you cannot use a private type for typing the formal parameter.

```
CLASS demo DEFINITION CREATE PRIVATE.
  PUBLIC SECTION.
    ...
  PROTECTED SECTION.
    ...
  PRIVATE SECTION.
    TYPES t_... TYPE ...
    METHODS constructor IMPORTING p TYPE t_...
ENDCLASS.

CLASS demo IMPLEMENTATION.
  METHOD constructor.
    ...
  ENDMETHOD.
    ...
ENDCLASS.
```

value is available even if the procedure ended with an error. For output parameters that are passed by value, the value is available only if the procedure ended without an error.

- `CHANGING` — An *input/output parameter* to which the value of an actual parameter is passed at the time of a call, and whose value is available in the actual parameter after the procedure is completed. For input/output parameters that are passed by reference, the value is available even if the procedure ended with an error. For input/output parameters that are passed by value, the new value is available only if the procedure ended without an error.

- `RETURNING` — A *return value* whose value is available in the actual parameter after the procedure is completed. A return value is always passed by value. It contains the result of a functional method that can have one return value and must have no output or input/output parameters.

• The way a formal parameter is **passed**

  - `REFERENCE` — When *passed by reference*, the procedure receives a reference to the actual parameter during the call and works with the actual parameter itself. Changes to the formal parameter are valid for the actual parameter immediately.

  - `VALUE` — When *passed by value*, a local data object is created as a copy of the actual parameter and the procedure receives the value of the actual parameter at calling time. Changed formal parameters are only passed to the actual parameter if the procedure was concluded without errors — that is, whenever the last statement is reached or it is left with `RETURN`.

- A formal parameter's **typing**
- If a formal parameter is **mandatory** or **optional**

Looking at all these options, you see that, compared to the more general rules for implementing a method, the definition of its parameter interface requires further, ABAP-specific, discussion.

> ***Note!***
>
> Besides the formal parameters listed here, the parameter interface of a procedure also contains the possible exceptions of the procedure. A parameter interface can contain either `RAISING` clauses for declaring the class-based exceptions that can leave the procedure or `EXCEPTION` clauses for defining the classic exceptions of the procedure. Dealing with exceptions will be covered in the guidelines for error handling in Part 3 of this article series.

Recommendation

When defining the parameter interface of a method (or a function module), keep the following points in mind:

- Regarding the **number** of formal parameters, as a rule the parameter interface of a procedure should be slim. Ideally, a method is functional — i.e., it has no or few importing parameters and one returning parameter.
- Regarding the **kind** of formal parameters, always use the appropriate kind. The kind of parameter defines the semantics of a parameter, while the method of passing is a more technical detail that should not be mixed up with the parameter's kind. For example, do not exploit the fact that an `EXPORTING` parameter passed by reference behaves like a `CHANGING` parameter. If you need an input/output parameter, define it as a `CHANGING` parameter and use `EXPORTING` parameters for output only (do not read it before it has first been written).
- For the way a parameter is **passed**, you must weigh performance vs. robustness. While passing by reference may provide better performance (especially for large parameters), passing by value is more robust. For example:
  - An `EXPORTING` parameter passed by reference is not initialized during the call (it behaves like a `CHANGING` parameter). Therefore, it is dangerous to read an `EXPORTING` parameter passed by reference before it has first been written. From the semantics of the parameter, you would expect it to be initial, but it contains the actual parameter's value. Also, when you insert lines into internal tables or concatenate contents to byte or character strings that are defined as such parameters, you must not forget to initialize the parameter first.
  - Another potential trap is that write access to `EXPORTING` and `CHANGING` parameters passed by reference works directly on the actual parameters. If the procedure is ended with an exception, the changes will not be undone — make sure that this is what you intend.
  - Even for `IMPORTING` parameters that are passed by reference, unexpected effects can occur if the procedure changes — either explicitly or implicitly — the actual parameter. In such an instance, the `IMPORTING` parameter will be changed even though it is protected from writing, from the method's point of view. This point is particularly problematic when passing system fields (see the sidebar "Passing system fields" on the next page).

For these reasons you must take special care when working with parameters passed by reference inside a procedure. Do not read `EXPORTING` parameters before they have been written (see the sidebar "Reading EXPORTING parameters" on

Passing system fields

In the example below, two methods with identical implementations are called. The first method (`m1`) works as expected by outputting the value of `sy-index` passed by value to `idx` three times — i.e., 1, 1, and 1 during the first invocation and 2, 2, and 2 during the second invocation. The second method (`m2`) outputs 1, 2, and 3 during both invocations. Here, the `DO` loop influences the global system field `sy-index` and hence `idx` (which references `sy-index`).

```
CLASS c1 DEFINITION.

  PUBLIC SECTION.
    CLASS-METHODS main.
  PRIVATE SECTION.
    CLASS-METHODS: m1 IMPORTING VALUE(idx)     TYPE i,
                   m2 IMPORTING REFERENCE(idx) TYPE i.

 ENDCLASS.

 CLASS c1 IMPLEMENTATION.

   METHOD main.
     DO 2 TIMES.
       m1( sy-index ).
       m2( sy-index ).
     ENDDO.
   ENDMETHOD.

   METHOD m1.
     DO 3 TIMES.
       WRITE / idx.
     ENDDO.
   ENDMETHOD.

   METHOD m2.
     DO 3 TIMES.
       WRITE / idx.
     ENDDO.
   ENDMETHOD.

 ENDCLASS.
```

page 14). Change an `EXPORTING` parameter or a `CHANGING` parameter before possible exceptions only if this is in accordance with the expected exception handling on the caller's side (exception handling will be covered in more detail in the guidelines for error handling in Part 3 of this article series), since even in the case of an exception the changed parameter will remain

Reading EXPORTING parameters

This example demonstrates a dangerous read access to the `EXPORTING` parameter `result`, which is passed by reference. The coding in the `IF` block is executed independent from the result of the `SELECT` statement, if the actual parameter passed by a caller is not initial. To clean up this example you must either clear `result` explicitly at the beginning of the procedure or — better — test `sy-subrc` instead of the target area in the `IF` statement.

```
CLASS demo DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS meth EXPORTING result TYPE dbtab.
ENDCLASS.

CLASS demo IMPLEMENTATION.
  METHOD meth.
    SELECT SINGLE *
           FROM dbtab
           INTO result
           WHERE ...
    IF result IS NOT INITIAL.
      ...
    ENDIF.
  ENDMETHOD.
ENDCLASS.
```

written. Do not pass system fields (or any other global data) by reference (more on this in the upcoming guidelines for processing data).

- For general hints regarding the **typing** of formal parameters, see the upcoming guidelines for declarations. Just one hint here: If you must define

Note!

For the dynamic ABAP types — strings and internal tables — the performance drawback of passing by value vs. passing by reference is mitigated somewhat by sharing,[11] which automatically takes place as long as neither the formal nor the actual parameter is changed.

[11] Sharing is a measure to improve the performance of assignments between dynamic data objects (*strings* and *internal tables*) even though dynamic data objects have value semantics and not reference semantics. When dynamic data objects are assigned or passed by value, the data is not copied immediately, only the required internal references for the string or the table body are transferred (for tables as of Release 6.10, and if the line type itself does not contain any table types). Sharing is not canceled until one of the involved data objects is changed. The actual copy process takes place only at this point (copy-on-write).

function modules, never type their parameters with the obsolete additions `LIKE` or `STRUCTURE`.

- Make **mandatory** only those parameters for which variable input is needed for each execution of the procedure. All other parameters should be **optional**. Otherwise, you force the caller to provide unnecessary actual parameters. Sometimes helper variables must be declared only for that purpose. Take care that all optional parameters are initialized appropriately. For dynamic initializations, you can use the predicate `IS SUPPLIED`. Don't use the obsolete predicate `IS REQUESTED` for this.

Defining control structures for processing blocks

Like most other programming languages, ABAP offers a variety of branches (`IF`, `CASE`) and loops (`DO`, `WHILE`). In addition to these general looping constructs, there are other, more ABAP-specific loops, such as those for internal tables (`LOOP`, `PROVIDE`) and in Open SQL (`SELECT`). Control structures are a means of structuring a processing block in statement blocks, which are structural units within control structures that consist of one or more consecutive statements. The statements of a statement block are processed sequentially. Statement blocks can in turn contain control structures. Possible control structures are sequence, branch, and loop. Control structures can be nested, in that closed statement blocks can in turn contain control structures.

Recommendation

For control structures we'll provide first some very general guidelines that could be given for other programming languages as well; we'll then extend that discussion with some ABAP-specific points.

General guidelines

- When defining control structures, never allow undefined behavior. Each `CASE/ENDCASE` block must contain a statement block following `WHEN OTHERS`. Each `IF/ELSEIF/ENDIF` chain must contain a final statement block following `ELSE`.
- If possible, follow the SESE (Single-Entry/Single-Exit) principle for control structures (not for procedures) — i.e., a control structure must be exited either normally or via a statement (`EXIT`, `RETURN`, `LEAVE`), but should not provide for exit using both techniques.
- Avoid excessive block nesting depth (e.g., `IF`, `LOOP`, `WHILE`). As a rule of thumb, the nesting depth level should not exceed a depth of three.

ABAP-specific guidelines

- Don't use the statements `CHECK` or `EXIT` outside of loops — to exit processing blocks use `RETURN` instead. `CHECK` and `EXIT` are context-sensitive; inside a loop they exit the loop, outside a loop they exit the current processing block. `RETURN` always exits the current processing block and is the dedicated statement for this purpose.
- Don't use the additions `VARYING/VARY` in `DO/WHILE` loops — become comfortable with the use of the modern alternatives `ASSIGN COMPONENT` and `ASSIGN INCREMENT` (see the sidebar "Accessing components in DO loops" on the next page). The reason is that `ASSIGN` and its additions offer a set of general functions for accessing sequences of memory areas that should be preferred, since they allow data manipulation "in-place" rather than requiring data movement. In this context, please note that the use of the addition `INCREMENT` with `ASSIGN` is dedicated for accessing sequences of equidistant memory areas that have the same data type. Other applications can result in runtime errors and in different behavior in Unicode and non-Unicode systems.[12]
- The short form `seltab` instead of the logical expression `dobj IN seltab` should be forbidden. The short form, where `seltab` must have been

[12] Remember the difference between a Unicode program and a Unicode system. A Unicode system is a single code-page system in which characters are encoded in Unicode character representation. While all ABAP programs on such an application server must be Unicode programs, Unicode programs can be executed in non-Unicode systems.

Accessing components in DO loops

Let's assume a structure `text` with at least four subsequent text-like components — `word1` to `word4` — that have the same length. The first `DO` loop accesses these components using the obsolete `VARYING` addition. Here, a data object `word` is used to access the contents of the components. If the structure `text` contains less than four of these components, the behavior can be undefined or even a runtime error might occur, if the range of `text` is exceeded.

```
DO 4 TIMES VARYING word FROM text-word1 NEXT text-word2.
  WRITE / word.
ENDDO.
```

The second `DO` loop shows how the functions of the first loop can be programmed more explicitly and more generically using the statement `ASSIGN INCREMENT`. Here, a field symbol `<word>` is used to point to the components instead of copying their contents to a variable. The success of the assignment can be checked in `sy-subrc` and the loop is exited when the memory range of `text` is exceeded.

```
DO 4 TIMES.
  idx = sy-index - 1.
  ASSIGN text-word1 INCREMENT idx TO <word> RANGE text.
  IF sy-subrc <> 0.
    EXIT.
  ENDIF.
  WRITE / <word>.
ENDDO.
```

For the second `DO` loop we assumed that the name but not the position of the first component in the series of four is known. If you know the position `pos` of the first component in the structure, you might also use the following `DO` loop using the statement `ASSIGN COMPONENT`.

```
DO 4 TIMES.
  idx = pos + sy-index - 1.
  ASSIGN COMPONENT idx OF STRUCTURE text TO <word>.
  IF sy-subrc <> 0.
    EXIT.
  ENDIF.
  WRITE / <word>.
ENDDO.
```

declared via `SELECT-OPTIONS seltab FOR dobj`, makes implicit use of a global data object `dobj` and is barely understandable to the reader of a program. This holds especially for methods, the only place where such operational statements should be implemented.

- Don't use the `LEAVE` statement without additions, because its behavior is context-sensitive and cannot be foreseen when coding the statement. Instead, use only the special purpose variants — `LEAVE PROGRAM` or `LEAVE SCREEN`, for example — because their behavior is clearly defined.

Now that you are armed with recommendations for using ABAP Objects, modularization, and program flow, let's turn our attention to declarations. We'll look at guidelines for this key aspect of ABAP programming next.

Best practices for declarations

It is a special strength of ABAP that you can define a great variety of data types and objects that stretch from very elementary types to very complex and dynamic types. Consequently, the subject of ABAP declarations is broad. It is beyond the scope of this article to tell you when and how to declare which data types. Some hints on types as they pertain to internal tables will be provided in the upcoming guidelines for processing data; in these following sections, however, we will restrict ourselves to some hints about:

- Type definitions
- Global data objects
- Declaration syntax
- Typing of formal parameters and field symbols

Type definitions

ABAP interprets the content of a data object in accordance with its data *type*. A data type is either the property of a data object (*bound data type*) or it can be defined as a *standalone* data type from which data objects can be instantiated.[13] Standalone data types can be defined:

- Globally
  - In the ABAP Dictionary
  - In type pools
  - In global classes or interfaces

[13] In this regard data types play the same role as classes. In fact, a data type is a kind of class without methods. In particular, note that data *types* and classes lie in the same name space while data *objects* have their own name space.

- Locally, using `TYPES`
  - In the global declaration part of an ABAP program
  - In local classes or interfaces
  - In procedures

The following recommendation handles how and where to define data types.

Recommendation

It is good programming style to construct a new data type only as a standalone data type and not as a bound data type. In this connection, it is important to note that the presence of a length specification in a `DATA` statement implies the construction of a new bound data type (see the sidebar "Constructing data types" on the next page). Constructing new data types as standalone types improves the maintainability of your ABAP programs because:

- A standalone type carries a semantic meaning.
- Such a type can be maintained centrally and reused at different places.
- The definition can be easily transferred to more general contexts if needed.

As to the question of *where* to define standalone data types, follow the defensive programming approach as introduced earlier in the guidelines for using ABAP Objects. That means you should define a data type in the most special context possible:

- Inside procedures, standalone types are normally only needed for helper variables whose technical settings are not defined by more general types of the same application.
- Global types (as of SAP NetWeaver '04) and constants can be defined in the public sections of global classes or in global interfaces. This allows you to create types and constants directly in the semantic context and hence automatically in the package (more on this in the guidelines for using

Constructing data types

The following line of code is a typical example, where the definition of a bound data type should be replaced by the definition and use of a standalone data type.

```
DATA word TYPE c LENGTH 10.
```

The above declaration can be replaced by the following if the data type is needed in the same context only. In most cases the data type will be defined in more general contexts.

```
TYPES word_10 TYPE c LENGTH 10.
DATA  word    TYPE word_10.
```

packages in Part 3 of this article series) where you need them. The declaration of public types and constants is the successor technique to the use of type pools. Therefore, do not create new type pools — use global classes or interfaces instead. Existing type pools (such as the system type pool ABAP, for example) can continue to be used, if needed.

- Consider carefully whether to declare global types in the ABAP Dictionary or in global classes or interfaces. For types in classes or interfaces, their use is bound to the use of their class or interface. Apart from their organization in a package (more on this in the guidelines for using packages in Part 3 of this article series), types declared in the ABAP Dictionary are somewhat free-floating. In order to avoid total swamping of the ABAP Dictionary with copious quantities of types that are needed for special purposes only, types in the ABAP Dictionary should be declared only when you really need the features that are offered by the dictionary — when defining types for Dynpro fields, for example (see the guidelines for GUI programming in Part 1 of this article series). Before defining a new data type, you should always consider whether you can reuse an existing data type — either one of your own or a general one. On the other hand, reuse only global types that exactly match your needs. Never reuse a non-technical data type — i.e., a data type with a semantic meaning — because of its technical settings only. In the same vein, try to avoid the definition of purely technical types but give your types a semantic meaning.

Global data objects

Each ABAP program (except type pools) implicitly has a global declaration part, following the opening statement of the program, which contains declarations that are visible in the implementation part of the program (see the guidelines for source code sequence in Part 1 of this article series). Besides data types, interfaces and classes, data objects, and field symbols can be declared there.

Recommendation

Global data — i.e., data objects declared in the global declaration part of a program — must be restricted to only those that are necessary for technical reasons. When programming with ABAP Objects and without using logical databases, the only global data that is necessary for technical reasons is the data that is needed for classic Dynpros. All other data must be declared in appropriate visibility sections of classes or as temporary working data in methods.

Other exceptions are global field symbols that are needed to address data objects that are referred by data reference variables with generic type (more on this in the guidelines for dynamic programming in

Part 3 of this article series). Data reference variables that are defined with `DATA dref TYPE REF TO data` cannot be de-referenced in operand positions with `dref->*`. The only way to access such data objects is by assigning `dref->*` to a field symbol. Since field symbols cannot be declared as components of classes, it can become quite cumbersome to declare helper field symbols in each method, especially if you use data references extensively. Therefore, it can be adequate to declare some appropriately typed field symbols (more on this in an upcoming section) in the global declaration part that are only used for de-referencing data reference variables.

> ***Note!***
>
> For all declarative coding that cannot be part of a class or a method, you should follow the stricter syntax rules of ABAP Objects (remember our fundamental rule from Part 1 of this article series, to use ABAP Objects wherever possible). This means all language elements that are forbidden in ABAP Objects, such as internal tables with header lines, should also not be declared outside of classes.

Declaration syntax

Regarding the syntax for defining types or declaring data objects, there are some ABAP specialties that deserve further discussion:

- When referring to an existing data type, you often have a choice between referring to a standalone data type using `TYPE` or referring to the bound data type of a data object of the same program (or of a public attribute of a global class) using `LIKE`.[14]
- When referring to a component of a structured data type, you can either refer directly to the data type of the component (`TYPE comp_type`) or to the name of the component (`TYPE struc-comp`).
- Additionally, even in Unicode-enabled classes, there are still some syntax variants for declarations that should no longer be used.

Recommendation

There are three areas to consider here:

- `TYPE` vs. `LIKE`
- Component type vs. type component
- Obsolete syntax variants

TYPE vs. LIKE

You should use the `LIKE` addition in all cases where your declaration is directly dependent on an existing data object or its components. An example is the declaration of helper variables in procedures that are closely related to the referenced data object — when using `LIKE LINE OF` for declaring a work area for an internal table, for example. As a result, your program remains valid even if the data type of the data object you are referring to is changed. In all other cases, you should use the `TYPE` addition to refer to the standalone data types directly.

Component type vs. type component

If you are only interested in the data type of the component, you should refer to it directly. If you want to work with the component of the structure, it is better to refer to the component since a change of its definition will not affect your application.

Obsolete syntax variants

The following list contains declaration syntax that should be considered obsolete:

- Following `VALUE` in `DATA`, `CONSTANTS`, etc., use only values that match the type or at least the value range of the declared data object. In

[14] But always remember that referring to database tables, flat structures, or views in the ABAP Dictionary using `LIKE` is obsolete and forbidden in ABAP Objects. Those dictionary objects are data types and must be addressed using `TYPE`.

particular, do not specify values that are too long. For non-matching values, a conversion takes place during program generation resulting in a value that may differ from the expectation raised by the coding.

- Do not type field symbols with the `STRUCTURE` addition of the `FIELD-SYMBOLS` statement. Field symbols declared using the addition `STRUCTURE` result in a mixture of typed field symbols and a utility for casting structured data types:

```
TYPES t_text TYPE c LENGTH 512.
DATA wa TYPE t_text.
FIELD-SYMBOLS <scarr> STRUCTURE
scarr DEFAULT wa.
<scarr>-carrid = '...'.
```

  Use the additions `TYPE` or `LIKE` for the `FIELD-SYMBOLS` statement to type field symbols, and use the addition `CASTING` of the `ASSIGN` statement for casting:

```
TYPES t_text TYPE c LENGTH 512.
DATA wa TYPE t_text.
FIELD-SYMBOLS <scarr> TYPE scarr.
ASSIGN wa TO <scarr> CASTING.
<scarr>-carrid = '...'.
```

- Do not use the `TABLES` statement except when declaring an interface to classic Dynpros (see the guidelines for GUI programming in Part 1 of this article series). For each Dynpro, declare a respective structure in the ABAP Dictionary that contains the definitions and texts for all fields of the Dynpro. Use an appropriate naming convention for that purpose. All other uses of `TABLES` are no longer required.

Typing of formal parameters and field symbols

The typing of formal parameters of procedures (see the earlier guidelines for defining the parameter interface of a procedure) and of field symbols (more on this in the guidelines for dynamic programming in Part 3 of this article series) is closely related to declarative statements, but also plays a special role. With the term *typing* we denote the declaration of a data type of a formal parameter or a field symbol. The special thing about typing is that a formal parameter and a field symbol receive their data type only at the moment when they are bound to an actual parameter during the invocation of a procedure, or bound to a data object during an assignment.

When assigning data objects to field symbols or passing actual parameters to procedures, the data type of the bound source field must match the type used for typing. Matching means that the technical type properties of the bound source field must be compatible to the typing of the formal parameter or field symbol. The technical type properties of an elementary data type are the built-in ABAP type (`c`, `d`, `f`, `i`, `n`, `p`, `t`, `string`, `x`, `xstring`), its length (for `c`, `n`, `p`, `x`), and its decimal places (for `p`). The technical type attribute of a structured type is its composition from sub-structures and elementary components, but the component names are not relevant. The technical type attributes of an internal table are its table category (`STANDARD`, `HASHED`, `SORTED`), its line type, and its table key.[15]

The typing of a formal parameter or a field symbol can be generic or complete. Besides serving as a basis for checking the assigned objects, the typing defines how a field symbol or formal parameter can be used as an operand. Statically, field symbols or interface parameters can be used in all operand positions that are not excluded by their type with one exception: as an operand representing a particular internal table, only a field symbol or a formal parameter with a matching table type is allowed.

For generic typing, ABAP offers a set of built-in generic types (`any`, `any table`, `c`, `clike`, `csequence`, `data`, `hashed table`, `index table`, `n`, `numeric`, `object`,[16] `simple`, `sorted table`, `standard table`,

[15] For reference variables, it makes no sense to define a compatibility that is related to technical type attributes. Instead, the static type and its relation to possible dynamic types must be considered.

[16] In fact, `object` is not really a generic type, but the root class of all classes in ABAP objects. However, since all kinds of object references can be passed to formal parameters or field symbols typed with `REF TO object`, `object` plays the same role for objects in this regard as `data` does for data objects.

`table`, `x`, `xsequence`) that can be used for the typing of field symbols and formal parameters of procedures. Furthermore, internal table types can be generic if the table key is not defined or is incomplete.

When you address field symbols or formal parameters that are typed generically, you must be aware of a little difference between static access and dynamic access:

- When you access a generically typed field symbol or formal parameter statically — i.e., by naming it or its components directly — the type properties of the data type used for typing are relevant.
- When you access a generically typed field symbol or formal parameter dynamically — i.e., by naming it or its components via the contents of a field — the type properties of the data type bound to the actual parameter are relevant.

Field symbols or formal parameters that are typed completely can be addressed like normal data objects. For static access as well as for dynamic access, the type properties of the data type used for typing are relevant.

Recommendation

Always type formal parameters and field symbols appropriately. For formal parameters, typing must match the needs of the implementation as well as the expectation of the user. If you want to type generically, be aware of the specific generic types `csequence`, `numeric`, `simple`, and `xsequence`. For example, use `csequence` for text processing instead of `any`. Similarly, use `standard table`, `sorted table`, `index table`, or `hashed table` instead of `any table`.

The more generally a field symbol or interface parameter has been typed, the more careful you must be when using it. Otherwise, you can quickly run into runtime errors. For example, assignments involving generically typed field symbols or parameters should be avoided if you cannot guarantee that the assignment will *always* be possible at runtime.

On the other hand, if you don't want to offer generic services, always type as specifically as possible, meaning always type completely. Only complete typing of formal parameters or field symbols can guarantee that your coding always behaves in the same way and that it can be tested locally. Be aware that for field symbols or formal parameters that are typed generically, not all type properties of an actual parameter are checked during binding (such as component names or semantic properties defined in the ABAP Dictionary, for example) and can be different from the typing of the formal parameter. Therefore, the behavior of static and dynamic access can be different (see the sidebar "Appropriately typing formal parameters" below).

Appropriately typing formal parameters

The following coding sequence exemplifies a little trap that you can stumble into if you do not type completely where it would be appropriate. In class `demo`, two table types are defined: `itabg` that is generic with respect to the table key, and `itabc` that is complete. The input parameter of method `sort_itabg` is typed generically with `itabg`, the input parameter of method `sort_itabc` is typed completely with `itabc`. In both methods the formal parameter `itab` is sorted statically as well as dynamically. The second `SORT` statement in `sort_itabg` results in a runtime error if the actual parameter `ltab` is passed as shown. Quite surprising at first sight! Why? According to the rules mentioned above, for a dynamic access of a generically typed formal parameter, the actual parameter's properties are relevant — and `ltab` has no component named `col2`! In method `sort_itabc` no error occurs, because the completely typed formal parameter has a component `col2` — independent from static or dynamic access.

Continues on next page

Continued from previous page

```
CLASS demo DEFINITION.
  PUBLIC SECTION.
    TYPES: BEGIN OF struc,
             col1 TYPE i,
             col2 TYPE i,
           END OF struc,
           itabg TYPE STANDARD TABLE OF struc,
           itabc TYPE STANDARD TABLE OF struc
                      WITH NON-UNIQUE KEY col1 col2.
    METHODS: main,
             sort_itabg IMPORTING value(itab) TYPE itabg,
             sort_itabc IMPORTING value(itab) TYPE itabc.
ENDCLASS.


CLASS demo IMPLEMENTATION.
  METHOD main.
    TYPES: BEGIN OF lstruc,
             col1 TYPE i,
             col3 TYPE i,
          END OF lstruc.
    DATA ltab TYPE STANDARD TABLE OF lstruc
                   WITH NON-UNIQUE KEY col1 col3.
    ...
    sort_itabg( ltab ).
    sort_itabc( ltab ).
  ENDMETHOD.
  METHOD sort_itabc.
     SORT itab BY col1 col2.
     SORT itab BY ('COL1') ('COL2').
  ENDMETHOD.
  METHOD sort_itabg.
     SORT itab BY col1 col2.
     SORT itab BY ('COL1') ('COL2'). "<- Runtime error!
  ENDMETHOD.
ENDCLASS.
```

You now have a powerful collection of guidelines at your disposal for using ABAP Objects, modularization, program flow, and declarations. Let's add another set of guidelines to your arsenal — best practices for processing data. We'll look at these next.

Best practices for processing data

The main task of most ABAP application programs is the processing of data. Data to be processed is stored as transient (data) objects in the internal session of an

ABAP program. The guidelines in the following sections cover:

- Using data objects
- Byte and character processing
- Using internal tables
- Using system fields

Using data objects

ABAP statements work with data objects in operand positions. The contents of data objects are treated according to the data object type. The data type also defines the value range. When the data type does not match the expected type of an operand position, in most cases the contents are converted to the expected type. You can also cast data objects to other data types.

Recommendation

The following list contains some recommendations for using data objects in operand positions:

- The data type of a data object must be chosen to match the expected values.
- Avoid overflow; use `TRY` constructs in critical cases.
- To avoid unnecessary conversions, inasmuch as possible, assign values only between data objects that have the same type.
- Avoid implicit casting. If necessary, an explicit cast with `ASSIGN CASTING` can be used. Situations where implicit casting occurs are assignments between incompatible structures or between elementary data objects and structures, comparisons between elementary data objects and structures, structures as operands where elementary types are expected, `SELECT * INTO wa`, and the use of the `INCREMENT` addition of the `ASSIGN` statement itself.
- Avoid magic numbers. Use constants instead of numeric literals. Exceptions are specifications such as `UP TO 1 ROWS`. Use constants proactively if possible — i.e., if you access a data object only for reading, use constants.
- Use text symbols instead of text literals or connect text literals to text symbols for all literals containing non-technical contents.
- If you use text literals without text symbols, those must not be relevant for translation. Only technical contents are allowed (e.g., to build a regular expression).
- Although ABAP allows invalid values within the value range of some elementary data types (`n`, `d`, `t`), assign only valid values to such data objects.
- If you must use character type fields to store numerical values, always place the sign in the leftmost position. There should be no gap between the sign and the number. The reason is that this mathematical notation is allowed for all conversions from character type fields to numeric fields. The commercial notation, where the sign is placed on the right, is not allowed for all numerical target types.
- For field symbols and reference variables, always assign data objects or references before using them. Trying to access (data) objects using initial field symbols or initial reference variables will lead to a runtime error.

Byte and character processing

For byte and character processing, ABAP generally distinguishes between two kinds of data types: text (type `c`) and byte (type `x`). Both have a static and a dynamic variant, namely fields and strings. So, we have text fields (type `c`), byte fields (type `x`), text strings (type `string`), and byte strings (type `xstring`). Other data types such as `n`, `d`, and `t` are also treated as text fields in most operand positions. Here are some basic facts about fields and strings:

- Text and byte fields have fixed lengths; strings have variable length. Strings are therefore more versatile and occupy only the memory really needed (besides the overhead required to manage the string, which consists of an internal reference and a header).

- The data type of text or byte fields is flat — they are stored in place. The data type of strings is deep, because they are handled internally via a reference. Text and byte fields can be accessed directly (e.g., with offset/length specifications) but the internal handling of strings is necessarily more complex than the direct handling of text and byte fields. On the other hand, strings are shared when assigned to other strings or passed by value. For performance reasons, no value copy takes place until the source or target variable is actually changed.
- Trailing blanks are ignored in most operand positions when assigning text fields, but they are retained when assigning text strings.
- A structure containing only text fields can be handled as one text field itself. A structure with text strings is deep and cannot be handled as a text field.

> ***Note!***
>
> When storing strings in database tables, you must distinguish between short and long strings (external data types `SSTRING` and `STRING`, respectively). Short strings are stored in place but are limited to a length of 255. The length of long strings is not limited, but they are not stored in place. There is a set of restrictions for using long strings in Open SQL statements. For example, database fields of type `STRING` cannot be addressed in `WHERE` conditions.

Recommendation

Except where a fixed length may be important (e.g., for screen fields with fixed layout), you should generally prefer the use of strings vs. text fields. The small internal administrative expense for strings is only relevant for short strings and is normally outweighed by sharing.

The accomplished ABAP developer should get acquainted with all ABAP statements for byte and character processing — don't try to invent work-arounds for existing language elements, such as `SPLIT` and `CONCATENATE`, for example:

- Use `FIND` instead of `SEARCH` and use only the new variant of `REPLACE` that is available as of Release 6.10. Both represent new developments and replace the older language elements. For example, only the new variants will allow you to use regular expression searches that are available with SAP NetWeaver 2004s.
- Where appropriate, use regular expressions (as of SAP NetWeaver 2004s) instead of programming your own pattern searches and replacements. Note, however, that while regular expressions are very powerful, they can also quickly become rather complex (and perhaps even defeat your attempts to make programs well-documented and readable). In order to keep your program readable, don't be so ambitious as to attempt to do everything with a single regular expression. If a regular expression becomes too complicated, don't hesitate to partition the work — with the help of explicit control structures, for example.
- Take advantage of the addition `ALL OCCURRENCES` in statements `FIND` and `REPLACE` (in `FIND` as of SAP NetWeaver 2004s) instead of using a `DO` or `WHILE` iteration for that purpose. For example, this `DO` loop replaces all occurrences of a character with another one in `text`:

```
DATA text TYPE string VALUE '1-2-3'.
DO.
  REPLACE '-' IN text WITH ','.
  IF sy-subrc <> 0. EXIT. ENDIF.
ENDDO.
```

  The addition `ALL OCCURRENCES` provides the same result:


```
DATA text TYPE string VALUE '1-2-3'.
REPLACE ALL OCCURRENCES OF '-' IN
text WITH ','.
```

Using internal tables

Internal tables are the most powerful data type in ABAP. They allow you to store a variable amount of data with fixed structure in memory. There are three kinds of internal tables:

- Standard tables
- Sorted tables
- Hashed tables

Internal tables can be accessed just like other data objects in general statements such as `MOVE` or via a set of dedicated statements that allow a wide variety of internal table operations. For accessing the lines of an internal table, there is a technical division:

- Accessing table lines using the table index — this method is available for standard and sorted tables and hence those are called *index tables*.[17]
- Accessing lines using the table key or a free search key — this method is available for all kinds of internal tables.

Accessing a line of an internal table via the table index is the fastest way to do so. Of course, for this to succeed the index must be known. Therefore, indexed accesses to standard tables or to sorted tables are generally preceded by a key access in order to determine the starting point.

Recommendation

Based on the kind of table you are processing, we recommend the following:

- **Standard tables** are ABAP's realization of dynamic arrays of lines having arbitrary (structured) types. The standard procedure for working with a standard table is:

  1. Fill the table completely with an Open SQL command or with `APPEND`.
  2. Sort the table according to your needs.
  3. Search for a starting point for sequential processing — e.g., by using `READ TABLE` with the `BINARY SEARCH` addition.
  4. Process the table sequentially using `LOOP`.
  5. Optionally, sort the table by some other sort key and start again with step 3.

  Since the costs for accessing table lines of standard tables via table key or another key is linear, dependent on the number of table entries, if you cannot benefit from `BINARY SEARCH`, you should not use (large) standard tables when key access to single lines is the most prevalent type of access.

- **Sorted tables** are appropriate if you need a fast key access as well as an index access and the items are to be sorted while the table is being filled. You fill a sorted table with statement `INSERT` according to the sort order defined by the table key. Access performance to a sorted table via table key has a logarithmic dependence on the number of table entries. Sorted tables are also suited for partly sequential processing in loops, if you specify only the beginning part of the table key in the `WHERE` condition of `LOOP`. This is also a valuable technique in connection with `READ`. Since sorted tables can be accessed using an index, they are also suited for applications where relations between tables are important. You can interlink

[17] Each table line in an index table has a unique line number, the so-called "table index," which is managed by the ABAP runtime environment. The line numbers start at 1 and run consecutively with no gaps, up to the current number of rows in the table. After accessing the line of an index table, the respective table index can be found in the system field `sy-tabix`. If necessary, the physical index is changed to a logical index, which causes an increased runtime demand. The logical index is created at precisely the moment in which a row is inserted before another row, the sequence of table rows is changed, or if any row (except for the last) is deleted. A logical index is not required as long as an internal table is exclusively filled with `APPEND` and if its bottom rows are exclusively deleted with `DELETE`.

internal tables in such a way that a table line of one table references a table line of another table by containing its index. Of course, the latter is also true for standard tables.

- **Hashed tables** are always appropriate when key access to table entries is the central operation. You cannot access hashed tables via an index. On the other hand, the cost for accessing a hashed table line via table key is constant and independent of the number of table entries. In the normal case, therefore, hashed tables have a better performance behavior than sorted tables for key accesses. The drawback compared to sorted tables is the greater memory overhead incurred by hashed tables and the constraint of requiring a unique key for each table line. When key access is the central operation, but the table key cannot be unique, you must use sorted tables instead of hashed tables.

From these points we can derive a general rule: sequential access to several lines is mainly the domain of standard and sorted tables, and access to single lines is mainly the domain of sorted and hashed tables.

Some general recommendations for internal tables:

- As long as all that you need from an internal table is to append lines to it and perhaps to sort it after filling it, standard tables are still the best choice. The other table types are too expensive for these simple tasks. Keep in mind that when inserting or deleting lines in index tables containing many lines, the administration of the logical index that manages the lines of the table internally can become expensive with regard to performance and additional memory space. Only for standard tables that are filled with `APPEND`, and where no lines are deleted except for the last line, is there no need for a logical index, and hence no additional costs are incurred.

- If memory space is an issue, for very large internal tables (greater than 500,000 lines) with a small line size, sorted tables might be preferable to hashed tables, since for internal administration they need only 6 bytes per line compared to 18 bytes for hashed tables.

Some special recommendations for statements for internal tables:

- When declaring internal tables, use the addition `INITIAL SIZE` only for inner tables in nested tables. For outer tables, the automatic allocation of initial memory size is appropriate. For inner tables, though, it may result in saving a large amount of memory.

- When reading internal tables with `READ TABLE` or `LOOP AT`, choose the appropriate output behavior. Writing into a work area `wa` with the addition `INTO wa` is only necessary if you want to change the work area without influencing the internal table. For pure reading purposes, or for modifying the contents of the internal table, the additions ASSIGNING <fs> for assigning internal table lines to a field symbol <fs> and `REFERENCE INTO dref` for setting a reference in a data reference variable `dref` to internal table lines are the better choices by far. See the guidelines for dynamic programming in Part 3 of this article series for when to use field symbols vs. reference variables.

- Use `CLEAR` instead of `REFRESH` for internal tables. The reason is that for internal tables without header lines, the general statement `CLEAR` works exactly the same as the special statement `REFRESH`. Since internal tables with header lines should no longer be used, the statement `REFRESH` is effectively obsolete. In order to free more memory, you can consider the use of `FREE`.

- The statement `COLLECT` should no longer be used for standard tables — use `COLLECT` mainly for hashed tables. The reason is that `COLLECT` is based on a hash algorithm. While the hash administration of a hashed table is always available and robust, for standard tables a temporary hash administration must be created when the `COLLECT` statement is used. This temporary hash administration is invalidated when the table is accessed for changing. If further `COLLECT` statements are entered after an invalidation, a linear search of all table rows must be performed. Furthermore, `COLLECT` only works properly on internal tables with unique lines, while a unique table key is guaranteed for hashed tables only. Therefore, for standard tables it cannot be guaranteed that the contents will always be suitable for editing using `COLLECT`.

- Don't use `APPEND SORTED BY` — use `SORT` instead. The reason is that creating a ranked list with `APPEND SORTED BY` is based on several assumptions about the internal table and how it is filled. Using `SORT` is the general method that can be applied to all kinds of tables and is independent from the mode of filling.

Using system fields

System fields are predefined, context-sensitive data objects that are filled by the ABAP runtime environment and can be utilized by an ABAP program. With one exception (`sy-repid`), system fields are variables.

Recommendation

Rules for the use of system fields:

- Write access to system fields should be strictly prohibited — overwriting such information could cause the loss of important information required for successful further program execution. The only system fields for which write access might conceivably be allowed (`sy-lsind`, `sy-lstat`, and `sy-tvar`) are part of classic list programming, which we have previously shown should not be used anyway.
- The use of system fields that are documented as obsolete or for internal use only is forbidden.
- If possible, evaluate system fields directly following the statements that set them. This holds especially true for `sy-subrc`.
- If a statement sets `sy-subrc`, evaluate it. If you don't want to do this for every statement execution in a productive system, you might use the statement `ASSERT` with external activation (as of Release 6.20, Support Package 29).
- Try to avoid system fields in a statement that influences those system fields, even when limiting your access to reading. Normally, the operand positions that are accessed for reading are evaluated first, but in some cases complex statements might influence the system fields before reading them.
- The direct use of system field types (structure `SYST` in the ABAP Dictionary) for defining input or output fields on the screen of a Dynpro is forbidden, because their semantic attributes (texts, documentation) are purely technical and were not foreseen for use on an application screen. For example, if you define an input field for a language specification it is not a good idea to refer to `SYST-LANGU`, because the documentation describes the meaning of `sy-langu` in a program and it is not suited for general language input.
- Do not use system fields as actual parameters. This is only critical if a parameter is passed by reference (see the earlier guidelines for defining the parameter interface of a procedure), but you can never be sure if the parameter interface of a procedure will not be changed in the future.

Conclusion

In this second part of our three-part article series, we have presented some best practices based on the fundamental rules and formal criteria we presented in the first part.

Let's conclude this second part by summarizing the most important best practices that are necessary to write up-to-date ABAP programs, from our point of view:

- Use ABAP Objects sensibly.
- Keep methods simple and usable.
- Be aware of the particular ABAP considerations when defining parameter interfaces of methods.
- Avoid control structures that are too complicated and may result in implicit behavior.

- Think carefully about where to declare data types and objects, and how to use them appropriately.

If you follow these practices in addition to the fundamentals we outlined in the first article, you will be well on your way toward better ABAP programming. The third and final part of this article series identifies further best practices and some administrative issues that will help you to reach that goal.

Acknowledgements

The authors want to express their gratitude to the following for their help in preparing this series of articles: Gerd Kluger, who contributed to the section on error handling, Jürgen Lehmann, who helped in the early preparation and participated in many discussions, and David F. Jenkins, who did a terrific job in editing this article and thereby helped to transform the first draft versions into something that could be published.

Andreas Blumenthal studied linguistics, mathematics, philosophy, and Jewish studies at the University of Heidelberg, Germany, where he received his Master's degree in 1985. After two years of research in the area of computer linguistics, he joined SAP in 1987. Andreas has participated in the R/3 project from the beginning. Working in the R/3 technology department, he has considerably contributed to the growth of ABAP into a modern, object-oriented programming language for business applications. Andreas became Development Manager of the "ABAP Language Group" in 1996. He is currently a Vice President and responsible for SAP NetWeaver Application Server ABAP.

Horst Keller holds a PhD in physics from the Technical University of Darmstadt, Germany. He joined SAP in 1995 after spending several years involved in research projects at various international institutions. As a Knowledge Architect within the SAP NetWeaver Application Server ABAP department, he is mainly responsible for the documentation and rollout of ABAP and ABAP Objects, while also developing the programs for formatting and presenting the ABAP documentation, including the related search algorithms. Horst is the author of the books "ABAP Objects – An Introduction to Programming SAP Applications" (Addison-Wesley Professional, 2002), "The Official ABAP Reference" (SAP PRESS, 2005), and the "ABAP Quick Reference" (SAP PRESS, 2005). Numerous other publications and workshops on this subject round off his work.

