# Build Flexibility and Control into Your ABAP Programs with Dynamic Programming Techniques

Holger Janz

*Holger Janz is a software developer at SAP AG in the Business Programming Languages Group. His responsibilities include parts of the ABAP Virtual Machine and the ABAP Compiler, and the integration of the JavaScript Virtual Machine into the SAP Web Application Server. His team is also responsible for the integration of Java into the SAP Web Application Server architecture.*

*(complete bio appears on page 96)*

Unlike languages that are intended primarily for building software tools, the ABAP programming language has evolved for 20+ years, driven by the needs of business application development. Consequently, as an SAP application developer, ABAP offers you some unique features that are not typically available in other languages. In particular, the concept of dynamic programming enables you to easily add runtime customization to your ABAP programs in several ways. Mastering these techniques will make your ABAP programs more powerful and flexible. And, it can reduce the need for excessive custom programming when you need to implement several features that are similar, but not identical.

This article explores all the key concepts that support dynamic programming within the SAP application server. We'll start by reviewing basic terminology and the concepts behind dynamic programming in general. We'll then move on to discussing and applying each of the six dynamic programming techniques that are provided in ABAP:

- Generic types
- Dynamic types
- Field symbols
- References
- Dynamic token specification
- Program generation

Along the way, you'll find plenty of examples and sample code to show you how to put each technique to work in your own programs.

These examples increase in complexity as the article progresses, with each building on the concepts illustrated in the previous one. For reasons of focus and space, this article assumes that you are already familiar with ABAP and ABAP Objects, as well as with both the runtime-event-oriented and object-oriented programming models.

Note that this article focuses primarily on features in Release 4.6 of the SAP application server, with references to key differences in later releases where applicable. At the end of the article, I'll summarize the enhancements in releases later than 4.6 that you will need to be aware of.

## Dynamic Programming Concepts and Benefits

What exactly is dynamic programming, and why is it so beneficial? Fundamentally, dynamic programming is used in almost every program, but the extent varies. Most programs consist of both dynamic and static parts. In this sense, programs are rarely black and white — most are shades of gray. Perhaps the best way to answer that question is to contrast dynamic with static programming.

In this context, *dynamic* and *static* apply to the data and/or operations performed on the data within a program. Static properties of data or operations are those characteristics that are known and fixed at compile time. They are explicitly defined in the program source code and will not change during runtime. In contrast, dynamic properties of data or operations are the characteristics that are *not* known or fixed at compile time. Instead, they are defined by the values of parameters that are passed to the program at runtime.

To understand what these characteristics mean in a practical context, let's review three examples that illustrate the static versus dynamic aspects of ABAP parameters and operations. In these examples, we will:

1. Calculate the number of days between two dates (a classic static subroutine)
2. Create a customized welcome message (a subroutine that is both static and dynamic)
3. Write any internal table to the ABAP List (a dynamic subroutine)

### *Example #1: A Classic Static Subroutine*

Consider a subroutine that calculates the number of days between two dates. It requires two dates as input parameters and produces a number of days as an output parameter. Based on this profile, you know that it has the following characteristics:

- The type and size of the input data (two dates) are known at compile time (static).
- The type and size of the output data (one number) are known at compile time (static).
- The operation of the subroutine (calculating the difference between two dates) is known at compile time (static).

If we consider only the subroutine operation, parameter types, and parameter sizes, we can classify this subroutine as static.

### *Example #2: A Subroutine That Is Both Static and Dynamic*

Next, consider a variation of the classic "Hello, World" programming exercise. This subroutine takes a person's name as an input parameter and produces a welcome message addressed to that person. Thus it has the following characteristics:

- The type of the input parameter (a person's name) is known at compile time (static), but its size (the length of the person's name) is known only at runtime (dynamic).

- The type of the output parameter (the welcome message) is known at compile time (static). However, its size is known only at runtime (dynamic) because the length of the resulting welcome message is affected by the length of the person's name.

- The operation of the subroutine is known at compile time (static): it must concatenate the welcome message and the person's name.

Here you see the first touch of dynamic programming. This subroutine is dynamic in terms of parameter size; it must be able to handle parameters of varying sizes.

### *Example #3: A Dynamic Subroutine*

In this final example, consider a subroutine that writes the contents of any internal table to the ABAP List.[1] This subroutine takes any internal table as an input parameter and writes all components of all lines to the ABAP List, which is the output parameter. Although this subroutine is more complex, you can see that it has the following characteristics:

- The type and size of the input parameter (any internal table) is known only at runtime (dynamic).

- Since the ABAP List is the output parameter, at compile time you don't know anything about its size (the count of lines and components) or the type of data that it will contain (dynamic).

- All subroutine operations are known only partially at compile time because you don't know anything about the count of `WRITE` statements needed or the type of data that will be written to the ABAP List (dynamic).

[1] Output pages that are created with `WRITE` statements are called ABAP Lists. They enable you to display data in a standard format based on type.

Here you see a subroutine that is dynamic in all three aspects — operations, parameter type, and parameter size. It must be able to handle internal tables of different types and sizes as input. Furthermore, it needs to be able to provide the right parameters for the `WRITE` statements for every component of every line as output.

### *Advantages and Disadvantages of Dynamic Programming*

Based on these three examples, we can now define dynamic programming as follows: Dynamic programming is the use of special programming constructs that enable you to use selected features of data and/or operations at runtime that cannot, for various reasons, be determined at compile time. While powerful, this flexibility has both advantages and disadvantages.

With the use of dynamic programming techniques, you can design your programs to be more flexible. For example, you can distinguish between parameter types at runtime and adapt both the behavior and sequence of operations. You can also write more generic subroutines instead of writing several specific subroutines. The third example illustrates the possibility of writing any internal table to the ABAP List. Thus you could use the same subroutine with a new internal table type, or even if an existing internal table type is changed.

However, this added flexibility isn't completely without cost. While programs and subroutines with dynamic features can be more flexible and generic, at the same time they also become more complex and harder to maintain. For example, the compiler cannot check or analyze dynamic features because the necessary information is not available until runtime. That means some checks must be done at runtime, which is not beneficial to performance. Furthermore, it is much harder to test a subroutine or program with dynamic features. The range of accepted parameters is much broader, and the runtime order of operations can depend on those parameters.

*Figure 1 Dynamic Programming Features in ABAP*

| Feature | Description | When to Use It |
|---|---|---|
| Generic type | Defines a group of similar (but not identical) data types. It is only a partial description of data; data cannot be created directly from this type. | For parameter typing, in order to handle different data types with similar features. |
| Dynamic type | A data type that can change its size at runtime. The ABAP runtime environment handles memory management. | If the data type is known at compile time, but the size is only known or changed at runtime. |
| Field symbol | An alias for data. | For dynamic access to datasets. |
| Reference | A safe pointer to data. | To construct and walk through networks of data. |
| Dynamic token specification | Allows parts of ABAP program statements to be specified at runtime. | For adapting the operation of the program based on data that is passed to it at runtime. |
| Program generation | Enables partial or full program generation at runtime. | As a last resort only, if the functionality or performance of other dynamic features is insufficient. |

Now that you understand the basic concepts of dynamic programming, it's time to look at how you implement them in ABAP. **Figure 1** gives you an overview of the ABAP dynamic programming features that we're about to discuss. Let's begin our exploration of dynamic programming techniques with the basic features of generic types.

## Generic Types

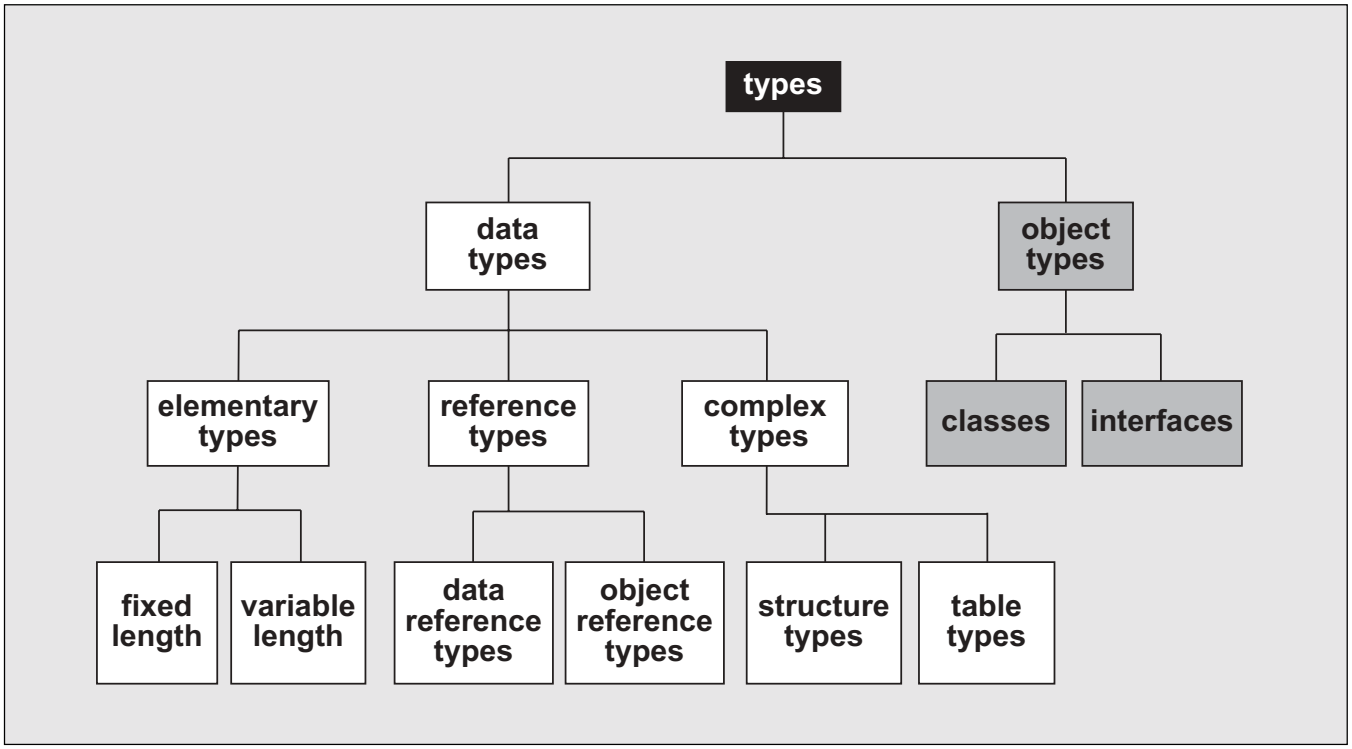
In order to understand generic types in the context of dynamic programming, first you need to understand the ABAP type hierarchy. ABAP is a hybrid programming language with a type system that reflects both the runtime-event-oriented and the object-oriented programming models. Object types (e.g., classes and interfaces) and value types (e.g., structures and tables) are integrated into one type system and share the same namespace. This means that a global class must not have the same name as a global structure type, and vice versa.

Think of types as building plans. Building plans provide descriptions of houses. Types are just descriptions of data. This analogy provides an easy way to distinguish between types and data. It is important not to confuse them.

A type is defined as "a set of properties that apply to all data of this type," while data is a sequence of bytes in the memory of the program. A type that contains the complete description required to create data is called a *concrete* type. In contrast, a *generic* type is only a partial description of data — for example, the length is not defined within the type. Since additional information is needed to fully describe the data, you cannot create data from a generic type. Generic types are used to define which properties a parameter must have in order to satisfy a

*Figure 2* *ABAP Type Hierarchy*




specific operation or subroutine. While data is always created from concrete types, data of several different types can make up a single generic type.

### *Data Types and Object Types*

As shown in **Figure 2**, ABAP types — whether generic or concrete — are categorized within a hierarchy that divides types into disjunctive subsets, data types and object types, which reflect ABAP's two programming models (runtime-event-oriented and object-oriented programming):

- **Data types** describe data objects. They are subdivided into elementary, reference, and complex types. Data types can be defined locally in the program or system-wide in the SAP application server repository. For example, integers, structures, and tables are all data types.
- **Object types** describe objects in terms of the object-oriented programming model. They are subdivided into classes and interfaces. Classes describe an object completely. Interfaces describe partial properties of an object.

ABAP contains several built-in elementary data types: **c**haracter (text), **n**umerical character, **d**ate, **t**ime, **i**nteger, **f**loating point number, **p**acked number, and byte code (he**x**adecimals). Some of these data types are concrete types (`D`, `T`, `I`, `F`). Other types (`C`, `N`, `P`, `X`) are generic, in the sense that when you create the data object, you must provide further information (e.g., length, or for type `P`, the number of decimal places).

In addition to the elementary data types `C` and `X`, which always refer to a fixed-length memory area at runtime, there are two corresponding data types with dynamic memory management. I'll discuss these in detail in the upcoming section on strings.

*Figure 3 Additional Generic Types in ABAP*

| Generic Type | Description | Supported in Release 4.6? |
|---|---|---|
| ANY | Includes all types | Yes |
| DATA | Includes all data types | Yes |
| ANY TABLE | Includes all table types | Yes |
| OBJECT | Includes all object types | Yes |
| NUMERIC | Includes the types I, P, and F | No (releases later than 4.6 only) |
| CSEQUENCE | Includes the types C and STRING | No (releases later than 4.6 only) |
| XSEQUENCE | Includes the types X and XSTRING | No (releases later than 4.6 only) |
| CLIKE | Includes all types of the generic type CSEQUENCE, plus the types N, D, and T | No (releases later than 4.6 only) |
| SIMPLE | Includes all types of the generic types NUMERIC, XSEQUENCE, and CLIKE | No (releases later than 4.6 only) |

As shown in **Figure 3**, ABAP also includes up to nine additional generic built-in types, depending on the release that you're running.

### *Creating Complex Data Types*

In addition to the elementary data types, ABAP provides four options for creating complex data types:

- Combine components of any data type to form a structure type.
- Use a line of any data type to form an internal table type.
- Combine attributes of any data type to form an object type.
- Use any type to form a reference type.

You can use structures and internal tables to create both complex data types and complex data objects. The table type, key, and uniqueness can be generic for table types, which means you can define a generic table type by omitting any of this information. However, you cannot create an internal table directly from a generic table type.

Both data types and object types can be used to form a reference type. A reference type is a data description that holds a reference to data. It simply points to other data, like a data signpost. In Release 4.6, only the generic type `DATA` can be used to form data reference types. Fully typed data references are only supported in releases later than 4.6.

### *Generic Types Overview*

In summary, ABAP supports three categories of generic types:

1. **Types without special properties:** `ANY`, `OBJECT`, and `DATA`. These types are used for dynamic operations or subroutines. No assumptions about the parameters can be made.

2. **Generic table types:** Table types are generic if one of the following properties is not defined: line type, key, table kind, or key uniqueness. No assumptions can be made for properties that are not defined.

3. **Types with generic length:** `C`, `N`, `X`, and `P`. No

*Figure 4 Comparison of Open SQL and ABAP Internal Table Operations*

| Open SQL Operation | Internal Table Operation |
|---|---|
| SELECT … | LOOP AT … ENDLOOP. |
| SELECT SINGLE … | READ … |
| INSERT … | INSERT … |
| UPDATE … | MODIFY … |
| DELETE … | DELETE … |

assumptions about length (number of characters or bytes) can be made at compile time.

Understanding how to use generic types is simpler when discussed in the context of how to use dynamic types. So, let's move on to a review of dynamic types, which includes examples that illustrate how to use generic types and how to handle undefined parameter properties.

## Dynamic Types

Dynamic types (specifically, internal tables and strings) are perhaps the most commonly used dynamic programming technique in ABAP. Almost every application developer has used one or more of them, even if unintentionally. Since the ABAP runtime environment handles memory management for dynamic types, you don't need to write any additional code for that purpose. As you will see, dynamic types are powerful development tools for accessing and manipulating data.

The difference between generic types and dynamic types is how each handles data properties. As you learned in the last section, generic types differ from concrete types because they omit some properties (e.g., length). With dynamic types, a property (e.g., length) is not omitted, but is instead variable at runtime. Thus you can create data directly from dynamic types, which isn't possible with generic types. When you do this, the ABAP runtime environment takes over the administration of the data property. For example, in the case of length, the data will grow and shrink automatically without any additional code.

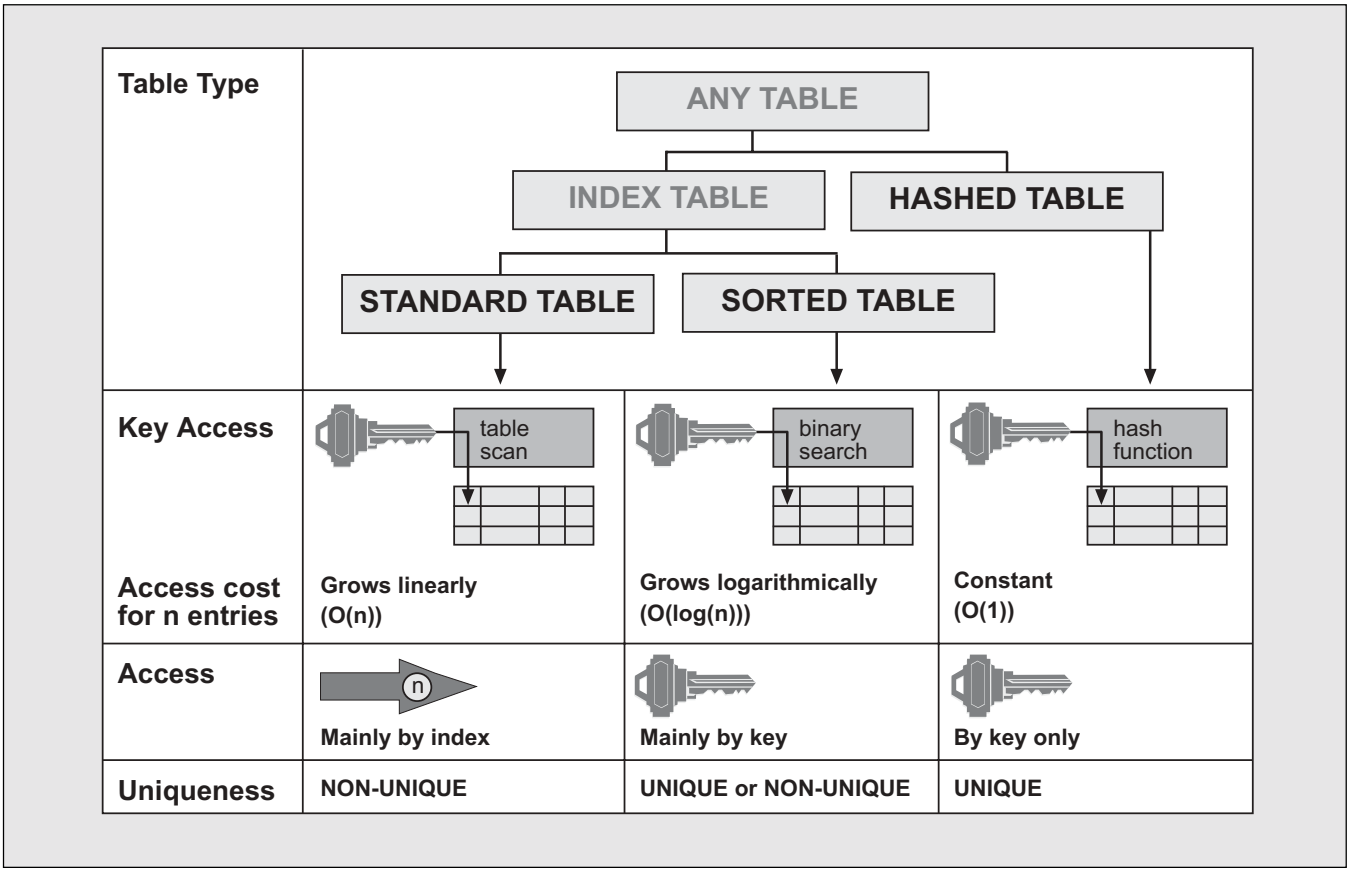
### *Internal Tables*

Internal tables are the most important dynamic data type of ABAP. They are useful for working with Open SQL, as well as for implementing other data types, such as stacks or dictionaries. Internal tables are a temporary representation of database tables. They are used to hold or to gather the results of database queries and to collect data for database update operations. In combination with Open SQL, internal tables are an important, effective, and stylish feature of ABAP for developing business applications.

As shown in **Figure 4**, ABAP supports operations on internal tables that are similar to those offered by database systems and Open SQL for database tables. Some Open SQL operations (such as subqueries, joins, etc.) are not directly supported by internal tables. However, you can simulate them with a combination of internal table operations.

ABAP supports three kinds of internal tables: standard tables, sorted tables, and hashed tables. Standard tables cannot have a unique key. The sequential order of `APPEND` and `INSERT` operations controls the order of table entries. Standard tables are mainly used for access by line number (`INDEX`). Sorted tables have a special sort key that is used for access, and the system guarantees the sort order after

*Figure 5 The ABAP Internal Table Type Hierarchy*




every operation. You can also access sorted tables by line number (`INDEX`). Hashed tables can only be accessed by key, but offer the fastest key access. Every hashed table has a key, and every key is unique.

**Figure 5** provides an overview of ABAP internal table types, including access method and associated cost. I introduced the generic table types `ANY` and `INDEX` in earlier sections.

A main feature of internal tables is that the ABAP runtime environment handles memory management. You, the developer, do not have to take care of it. A closer look at internal tables also shows that they can be more than just tables. From a technical or abstract point of view, they are like an array or a double-linked list. With this knowledge, you can use internal tables to implement other abstract data types.

**Figure 6** contains an example of using an internal standard table (which behaves like an array) to implement an integer stack. The `LCL_INTEGER_STACK` class encapsulates the implementation of an integer stack with an internal table. The private attribute `STACK` is an internal standard table, which can only be accessed by the methods `PUSH`, `POP`, and `WRITE`. The method `PUSH` inserts an integer `P_X` at `INDEX 1`. If `STACK` contains any lines, all line numbers will be incremented by one (1 becomes 2, 2 becomes 3, etc.) and `P_X` will be inserted as the new number 1. The method `POP` does the inverse. It reads the first line into `P_X` and deletes it. If the read operation fails, the exception `STACK_UNDERFLOW` is raised. This is only possible if the internal table `STACK` is empty. The method `WRITE` loops over the internal table `STACK` and writes the value of every line.

*Figure 6 Using Internal Standard Tables to Implement an Integer Stack*

```
PROGRAM stack_example.

CLASS lcl_integer_stack DEFINITION.
  PUBLIC SECTION.
    METHODS:
      push IMPORTING p_x TYPE i,
      pop RETURNING value(p_x) TYPE i EXCEPTIONS stack_underflow,
      write.
  PRIVATE SECTION.
    DATA stack TYPE STANDARD TABLE OF i.
ENDCLASS.

CLASS lcl_integer_stack IMPLEMENTATION.

  METHOD push.
    INSERT p_x INTO stack INDEX 1.
  ENDMETHOD.

  METHOD pop.
    READ TABLE stack INTO p_x INDEX 1.
    IF sy-subrc <> 0.
      RAISE stack_underflow.
    ENDIF.
    DELETE stack INDEX 1.
  ENDMETHOD.


  METHOD write.
    DATA x TYPE i.
    WRITE / 'stack = {'.
    LOOP AT stack INTO x.
      WRITE x.
    ENDLOOP.
    WRITE '}'.
  ENDMETHOD.
ENDCLASS.

DATA:
  my_stack TYPE REF TO lcl_integer_stack,
  x TYPE i.

START-OF-SELECTION.
  CREATE OBJECT my_stack.

  CALL METHOD my_stack->push EXPORTING p_x = 5.
  CALL METHOD my_stack->push EXPORTING p_x = 7.
  CALL METHOD my_stack->push EXPORTING p_x = 11.
  CALL METHOD my_stack->push EXPORTING p_x = 13.
  CALL METHOD my_stack->pop  RECEIVING p_x = x.
  CALL METHOD my_stack->push EXPORTING p_x = 17.
  CALL METHOD my_stack->write.
```

Define the integer stack class with the PUSH, POP, and WRITE methods and the private standard table STACK.

The PUSH method inserts a value at the beginning of the table.

The POP method starts reading from the beginning of the table. If an entry is not found, it raises an exception. Otherwise, it deletes the entry and returns the value.

The WRITE method loops over the table and writes the value to the ABAP List.

Declare a reference to an instance of the integer stack.

Create an instance of the integer stack and move a reference into MY_STACK.

First PUSH 5, 7, 11, and 13 to the integer stack, then POP 13 from the integer stack and move it into variable X. Then PUSH 17 to the integer stack. At the end, write the integer stack to the ABAP List.

*Figure 7 Modifying an Internal Table Within a Loop*

```
PROGRAM delete_in_loop_example.

DATA:
  wa TYPE i,
  itab TYPE STANDARD TABLE OF i.

APPEND 1 TO itab.
APPEND 2 TO itab.
APPEND 3 TO itab.

LOOP AT itab INTO wa.
  IF sy-tabix = 2.
    DELETE itab INDEX 2.
  ENDIF.
ENDLOOP.

LOOP AT itab INTO wa.
  WRITE / wa.
ENDLOOP.
```

Declare the integer table and work area.

Fill the integer table with dummy data.

Loop over the integer table and delete the entry at index 2.

Write the resulting integer table to the ABAP List.

In Figure 6, you saw how easy it is to use an internal standard table to implement a stack. However, working with internal tables can sometimes be tricky.

For example, consider what would happen if an internal table is modified within a loop over the same table. **Figure 7** shows a situation where the result might surprise you. The internal table `ITAB` is filled with three entries. In the first loop, the second line is deleted at `INDEX 2`. In the second loop, every line of the internal table is written. The resulting list only contains the number 1. Why did this happen? In the second pass of the first loop, the third line becomes the second. In the third pass, the new second line is visited. In this example, think of an internal table as a double-linked list where every entry is visited exactly once. Because of the deletion, the indices of the table lines are changing, but every line will be visited.

In a later example, I'll show you how to use a hashed table, along with other features, to implement a dictionary. But first we need to examine other features that will be needed for that exercise. Let's start with strings, which are another dynamic ABAP data type.

### *Strings*

A string is a new and powerful ABAP data type that was introduced with Release 4.6. It is a very elegant way to handle character sequences. ABAP supports two kinds of strings: character strings `STRING` and byte strings `XSTRINGS`. Think of them as arrays with a variable number of characters or bytes. As for internal tables, memory management for strings is handled by the ABAP runtime environment. For example, you do not have to do anything with memory management if the size of a string is changed during an operation.

Additional string features include "copy on write" and "sharing," which are also handled by the ABAP runtime environment. You can use these two features for building applications without writing any additional code:

- **Copy on write** means that a string is not copied on assignment, but only during a write operation. The advantage is that string assignment is very cheap. In releases later than 4.6, copy on write is also implemented for internal tables.

- **Sharing** is used when converting ABAP character fields to ABAP strings. At first it might sound strange that you must convert an ABAP character field into an ABAP character string. However, remember that ABAP character fields are blank-padded (i.e., padded with spaces). During all operations on character fields, either trailing spaces are ignored or spaces are appended to fill the length of the resulting character field. Based on this behavior of character fields, you can emulate character strings with a maximum length. However, character strings do not have a maximum length; they are limited only by the amount of available memory. The length of a string is always the actual number of characters to be used by all operations. Trailing spaces of strings are recognized.

For assignment, the same rules apply to fields and strings as to all other operations. The trailing spaces of a character field are ignored when assigned to a character string. Thus a space (a one-character field containing " ") results in an empty string (a string of actual length zero). To use trailing spaces with strings, you need to use special string constants that are implemented in releases later than 4.6. For example, in Release 4.6 there is a useful trick to generate a character string that contains one space. I cover this in the next example.

All character field operations are supported for character strings. You can also mix character fields and strings in any operation requiring more than one parameter.

The example in **Figure 8** shows how easy it is to use strings without dealing with length or memory management. It uses strings to split a sentence into words and then reconstruct the sentence.

*Figure 8 Using Strings to Manipulate Data*

```
PROGRAM string_example.

PARAMETER:
  sentence(255) TYPE c LOWER CASE DEFAULT 'This is a test!'.

DATA:
  str TYPE string,
  str_space TYPE string,
  str_tab TYPE TABLE OF string,
  str_wa TYPE string.

SHIFT str_space RIGHT.


str = sentence.
WRITE: / str, /.

SPLIT str AT '' INTO TABLE str_tab.
LOOP AT str_tab INTO str_wa. "no string copy at this assignment
  WRITE / str_wa.
ENDLOOP.
```

Declare the input parameter.

In Release 4.6, use this trick to create a string with one space. Special constants are available in later releases.

First write the sentence as is.

Split the sentence into words in the string table STR_TAB. Write each word on a separate line.

*(continued on next page)*

*Figure 8 (continued)*

```
WRITE /.

CLEAR str.
LOOP AT str_tab INTO str_wa. "no string copy at this assignment
  CONCATENATE str str_wa INTO str SEPARATED BY str_space.
ENDLOOP.
SHIFT str LEFT.
WRITE / str.
```

Reconstruct the sentence by concatenating the words of the string table STR_TAB, separated by spaces.

> ✓ ***Tip***
>
> *Whenever the length of a character field is variable, use ABAP character strings instead of ABAP character fields. Only use ABAP character fields for fixed length character fields. Note that using ABAP strings with Open SQL and in database tables is only supported in releases later than 4.6.*

The program in Figure 8 takes a sentence as parameter SENTENCE. First, the character string STR_SPACE containing one space is created. Note that the SHIFT operation is the *only* way to do it in Release 4.6.

Next, SENTENCE is assigned to the character string STR, ignoring trailing spaces. The string STR is written to the ABAP List. Now the string STR is split into words in table STR_TAB, which is a standard table with a line type of STRING. After the sentence is split, each word is written to the list on a separate line. No string copy is done, because no strings in table STR_TAB are changed in the first loop.

At the end, the original string is reconstructed and written to the ABAP List.

As the example in Figure 8 shows, using strings is easy and intuitive (if we forget for one moment the lack of support for string constants in Release 4.6). However, you have to keep in mind the difference between character fields and character strings (even very small ones!).

Now let's look at using field symbols and references in ABAP. When used with the data types we've already discussed, they provide very safe and robust methods of accessing data dynamically.

## Field Symbols

Field symbols are a special feature of ABAP not found in most other programming languages. Their primary purpose is for accessing lines of internal tables, copy-free. Think of field symbols like name aliases for existing data. You can assign a new name to existing data (or part of it), and work on that data under its new name. Field symbols can be typed with generic types.

Before using a field symbol, you must first declare it and give it a type. Its name must begin with "<" and end with ">". The syntax to declare a field symbol looks like the syntax to declare a variable:

```
FIELD-SYMBOLS <name> TYPE typename.
```

You must also assign the field symbol to existing

data. Not assigning a field symbol is like not naming a person. You can use various ABAP statements to assign a field symbol. Here we'll discuss the two most useful statements:

- ASSIGN
- LOOP-ASSIGNING

The ASSIGN statement assigns a field symbol to any data:

```
ASSIGN <field_symbol_name> TO
data_name.
```

As for parameters that are passed to a subroutine, a type check is performed between the field symbol type and the data type. The operation is executed only if the types match.

There are several other variants of the ASSIGN statement, such as for assigning a component of a structure. For example, the LOOP-ASSIGNING statement assigns a field symbol to a table line:

```
LOOP … ASSIGNING <field_symbol_name>.
…
ENDLOOP.
```

Instead of copying the whole table line into a work area, the field symbol is assigned. Assigning a field symbol is cheaper than copying the table line, especially for tables with long lines.

There are also special statements to explicitly unassign a field symbol, or to check whether a field symbol is assigned:

```
UNASSIGN <field_symbol_name>.

… <field_symbol_name> IS ASSIGNED …
```

See **Figure 9** for an example of how to use field symbols. It uses a combination of field symbols and generic types to implement a list writer that you can use as an output method for any internal table.

The class LCL_UTIL is used to implement the table list writer. The method WRITE_TABLE implements the list writer for tables. This method has exactly one parameter to pass to. Because the parameter P_TABLE is typed with the generic type ANY TABLE, it can pass any table to this method.

*Figure 9 Using Field Symbols and Generic Types to Implement a List Writer*

```
PROGRAM field_symbols_example.

CLASS lcl_util DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      Write_table IMPORTING p_table TYPE ANY TABLE.
ENDCLASS.

CLASS lcl_util IMPLEMENTATION.
  METHOD write_table.
    FIELD-SYMBOLS:
      <line> TYPE ANY,
      <field> TYPE ANY.
    WRITE / '{'.
```

Declare the class LCL_UTIL with the WRITE_TABLE method.

Implement the class LCL_UTIL with the WRITE_TABLE method.

*(continued on next page)*

*Figure 9 (continued)*

```
    LOOP AT p_table ASSIGNING <line>.
      WRITE /4 '('.
      DO.
        ASSIGN COMPONENT sy-index OF STRUCTURE <line> TO <field>.
        IF sy-subrc <> 0.
          EXIT.
        ENDIF.
        WRITE /8 <field>.
      ENDDO.
      WRITE /4 ')'.
    ENDLOOP.
    WRITE / '}'.
  ENDMETHOD.
ENDCLASS.

TYPES:
  BEGIN OF my_line_type,
    name TYPE string,
    birth_date TYPE d,
    salary TYPE p DECIMALS 2,
  END OF my_line_type,
  my_table_type TYPE HASHED TABLE OF my_line_type
    WITH UNIQUE KEY name birth_date.

DATA:
  my_wa TYPE my_line_type,
  my_table TYPE my_table_type.

START-OF-SELECTION.

  my_wa-name = 'Peter Smith'.
  my_wa-birth_date = '19640303'.
  my_wa-salary = '20000.00'.
  INSERT my_wa INTO TABLE my_table.

  my_wa-name = 'April May June'.
  my_wa-birth_date = '19660401'.
  my_wa-salary = '1000000.00'.
  INSERT my_wa INTO TABLE my_table.

  my_wa-name = 'Mary McWeather'.
  my_wa-birth_date = '19710711'.
  my_wa-salary = '1.00'.
  INSERT my_wa INTO TABLE my_table.

  CALL METHOD lcl_util=>write_table EXPORTING p_table = my_table.
```

Loop over all components of all lines of the table P_TABLE and output the value to the ABAP List using field symbols.

Declare the example table MY_TABLE and fill it with data.

Output the example table to the ABAP List with the WRITE_TABLE method of the class LCL_UTIL.

The method WRITE_TABLE uses two field symbols. Because both field symbols are typed with ANY, they can be assigned to data of any type. A LOOP-ASSIGNING statement assigns the field symbol <LINE> and loops over the table passed by the parameter P_TABLE. In the LOOP, the field symbol <FIELD> is assigned by a special variant of the ASSIGN statement. Here it assigns the component of a structure based on its position, but it works with component names, too. All components of a structure are enumerated from 1 to n. SY-INDEX is the loop count of the DO-ENDDO loop. The ASSIGN statement sets the SY-SUBRC to not equal 0 if SY-INDEX is greater than the component count of the structure that was passed.

You would need to modify this example to handle nested tables correctly. The type of the field symbol <FIELD> must be checked after every assignment. If <FIELD> is a table, the method must be called recursively. I'll explore the required techniques later in this article.

As you can see, field symbols are a safe method for looping over internal tables. This method is also faster than using a work area (especially if the table line is very long) because the whole table line is not copied to a work area. Furthermore, you can also use field symbols to access components of structures dynamically.

However, remember that a field symbol is simply an alias for data and all the operations that work on it. Next, I'll discuss references, which are similar in concept but offer more powerful options for accessing data.

## References

References are a common feature supported by most programming languages, and ABAP is no exception. Like field symbols, references must be typed. However, the difference between field symbols and references is that a reference is not merely an alias for data. A reference is actually data that points to other data. You can use references to produce data chains and recursions between data. Furthermore, you can define references as parts of structures, objects, and table lines. These constructs are not possible with field symbols.

ABAP supports two kinds of references:

- **Object references** point to ABAP Objects. An object reference is either a class or an interface reference. (Since these references are fundamental concepts of object-oriented languages, they are not discussed in this article.)

- **Data references** point to data. Note that typed data references are only supported in releases later than 4.6.

Both object and data references are actually data themselves, even though they are simply pointers to other data. All references are the same size and, like any data, must be declared as follows:

```
DATA ref_name TYPE REF TO type_name.
```

The newly declared data reference is considered to be in an "initial" state and does not yet point to any data. As with any data, you can check the state of a data reference with the operator IS INITIAL. Once declared, you can fill a data reference in one of three ways:

- Create a reference for existing data using the statement GET REFERENCE:

```
GET REFERENCE OF data_name INTO
ref_name
```

- Get a reference for new data created at runtime using the statement CREATE DATA:

```
CREATE DATA ref_name TYPE type_name.
```

- Copy a reference using the operator "=":

```
ref_name = ref_name2.
```

*Figure 10 Using References to Access and Manipulate Data*

```
PROGRAM data_ref_46.

DATA data_ref TYPE REF TO data.
FIELD-SYMBOLS <fs> TYPE i.

CREATE DATA data_ref TYPE i.

ASSIGN data_ref->* TO <fs>.
<fs> = 13.

ASSIGN data_ref->* TO <fs>.
WRITE / <fs>.

CLEAR data_ref.
```

Declare a reference and a field symbol.

Create an integer at runtime and move the reference to it to DATA_REF.

Access the dynamically created integer via the field symbol <FS>, then change and output the integer pointed to by the reference.

Clear the reference to allow the ABAP runtime environment to end the use of the data.

All data reference operations affect the reference, rather than the data that it points to. Thus a move between two references will move only the reference; after the move, both references would point to the same data. In order to access the data itself, you need to use the "`->*`" special operator. Note that in Release 4.6, this operator is only implemented for the `ASSIGN` statement. For all other operations in Release 4.6, you cannot access the data that is pointed to by a reference in a single step. Instead, you need to use a field symbol.

To get a better idea of how to work with references, let's look at an example of references in action. **Figure 10** shows how to use references to create, write, and read data in Release 4.6. Note that in releases later than 4.6, you can use the "`->*`" operator in any statement (not just in the `ASSIGN` statement).

Note the use of the `CREATE DATA` statement in this example. When you use this statement, any data that is created at runtime will be handled by the ABAP Objects garbage collection mechanism. You do not need to write any code to explicitly destroy the data or handle memory management. Instead, the garbage collection mechanism handles both tasks automatically, as long as the data is no longer used by any references. While you don't need to add any code to free up memory, you do need to clean up any references. As long as even one reference points to the data, the ABAP runtime environment cannot free the memory. As shown at the end of the example in Figure 10, the `CLEAR` statement ends the use of the data by a reference.

Now let's look at a practical (although more complex) example of how you might use references. **Figure 11** shows you how to combine strings and references to implement a dictionary. A dictionary in this context is an abstract data structure for storing and retrieving data with a unique key. Data is stored and retrieved with the same string key. References can be used to point to any amount of data (e.g., nested internal tables and structures). The key of any dictionary entry is unique.

Here, the class `LCL_DICTIONARY` is used to implement a dictionary for storing data references and string keys. It has four methods: `PUT`, `GET`, `REMOVE`, and `WRITE`. The `PUT` method takes a string and a data reference as input parameters. It inserts the entry into the dictionary `DICTIONARY`, which is stored in a hashed table with a string as

*Figure 11 Using Strings and References to Implement a Dictionary*

```
PROGRAM dictionary_example.

CLASS lcl_dictionary DEFINITION.
  PUBLIC SECTION.
    METHODS:
      put
        IMPORTING p_key TYPE string p_value TYPE REF TO data
        EXCEPTIONS duplicated_key,
      get
        IMPORTING p_key TYPE string
        EXPORTING p_value TYPE REF TO data
        EXCEPTIONS key_not_found,
      remove
        IMPORTING p_key TYPE string
        EXCEPTIONS key_not_found,
      write.
  PRIVATE SECTION.
    TYPES:
      BEGIN OF dictionary_line_type,
        key TYPE string,
        value TYPE REF TO data,
      END OF dictionary_line_type,
      dictionary_type TYPE HASHED TABLE OF dictionary_line_type
        WITH UNIQUE KEY KEY.
    DATA:
      dictionary TYPE dictionary_type.
ENDCLASS.

CLASS lcl_dictionary IMPLEMENTATION.
  METHOD put.
    DATA x TYPE dictionary_line_type.
    x-key = p_key.
    x-value = p_value.
    INSERT x INTO TABLE dictionary.
    IF sy-subrc <> 0.
      RAISE duplicated_key.
    ENDIF.
  ENDMETHOD.
  METHOD get.
    DATA x TYPE dictionary_line_type.
    READ TABLE dictionary INTO x WITH TABLE KEY key = p_key.
    IF sy-subrc <> 0.
      RAISE key_not_found.
    ENDIF.
    p_value = x-value.
  ENDMETHOD.
  METHOD remove.
    DELETE TABLE dictionary WITH TABLE KEY key = p_key.
    IF sy-subrc <> 0.
```

Declare a dictionary class with the following methods:
- PUT to store a data reference with a string key
- GET to retrieve a data reference by string key
- REMOVE to remove a data reference from the dictionary by string key
- WRITE to write the dictionary to the ABAP List

Note the private declaration of a dictionary class. The dictionary is actually implemented with the internal table DICTIONARY.

Implement the PUT, GET, REMOVE, and WRITE methods of the dictionary class. Use the internal table operation on the internal table DICTIONARY, which is used to hold the dictionary data.

*(continued on next page)*

*Figure 11 (continued)*

```
      RAISE key_not_found.
    ENDIF.
  ENDMETHOD.
  METHOD write.
    DATA x TYPE dictionary_line_type.
    FIELD-SYMBOLS <fs> TYPE ANY.
    WRITE / 'dictionary = {'.
    LOOP AT dictionary INTO x.
      ASSIGN x-value->* TO <fs>.
      WRITE: /4 x-key, ':', <fs>.
    ENDLOOP.
    WRITE / '}'.
  ENDMETHOD.
ENDCLASS.


DATA:
  my_dictionary TYPE REF TO lcl_dictionary,
  data_ref TYPE REF TO data.

FIELD-SYMBOLS:
  <fs> TYPE ANY.

START-OF-SELECTION.
  CREATE OBJECT my_dictionary.

  CREATE DATA data_ref TYPE i.
  ASSIGN data_ref->* TO <fs>.
  <fs> = 13.
  CALL METHOD my_dictionary->put
    EXPORTING p_key = 'Integer' p_value = data_ref.

  CREATE DATA data_ref TYPE f.
  ASSIGN data_ref->* TO <fs>.
  <fs> = '13.13'.
  CALL METHOD my_dictionary->put
    EXPORTING p_key = 'Float' p_value = data_ref.

  CREATE DATA data_ref LIKE sy-repid.
  ASSIGN data_ref->* TO <fs>.
  <fs> = sy-repid.
  CALL METHOD my_dictionary->put
    EXPORTING p_key = 'Very Very Very Long Key'
              p_value = data_ref.

  CALL METHOD my_dictionary->get
    EXPORTING p_key = 'Float'
    IMPORTING p_value = data_ref.
  ASSIGN data_ref->* TO <fs>.
  WRITE: / 'Float :', <fs>.
```

Declare and create the dictionary MY_DICTIONARY.

Fill the dictionary with a reference to an integer of value 13 with the string key “Integer”.

Fill the dictionary with a reference to a float of value 13.13 with the string key “Float”.

Fill the dictionary with a reference to a character field containing the program name with the string key “Very Very Very Long Key”.

Retrieve the dictionary entry with the string key “Float” and write the data pointed to by the reference to the ABAP List.

*Figure 11 (continued)*

```
CALL METHOD my_dictionary->remove
  EXPORTING p_key = 'Float'.

CALL METHOD my_dictionary->write.
```

Remove the dictionary entry that contains the string key "Float".

Write the dictionary to the ABAP List.

its key. The `GET` method takes a string and the key of an entry in the dictionary as input parameters, and returns the corresponding data reference. The `REMOVE` method takes a string as input and removes the corresponding entry from the dictionary. The `WRITE` method writes a dictionary to the ABAP List.

In the example in Figure 11, the dictionary `MY_DICTIONARY` is created and three entries are inserted:

1. A data reference to an integer number, with the value `13` and the dictionary key `Integer`.
2. A data reference to a floating point number, with the value `13.13` and the dictionary key `Float`.
3. A data reference to a character field, with the program name as the value and the dictionary key `Very Very Very Long Key`.

Note that the reference with the key value `Float` is retrieved from the dictionary with the method `GET`. The data pointed to by the reference is written to the list and the entry `Float` is removed from the dictionary. At the end of this example, the dictionary is written to the ABAP List.

Up until now, we have looked at various ways of using generic and dynamic data types in ABAP to *access* data dynamically. Now we'll begin looking at ways of *generating* ABAP statements dynamically, which enables data operations to be performed dynamically. Certainly we aren't overlooking inheritance and interfaces, which are fundamental characteristics of any object-oriented programming language and thus not covered specifically in this article. However, ABAP also provides another simple mechanism to execute operations dynamically where needed — dynamic token specification.

## Dynamic Token Specification

Most ABAP statements allow you to specify some part of the statement dynamically. Essentially, this means you can supply various components of ABAP statements at runtime with a common syntax.

The best way to understand this concept is via an example: Suppose you have an internal table, which must have a specified sort order. The name of the component to be used for sorting the internal table could be specified at runtime and stored in either a character field or string. Dynamic token specification is the technique you would use to meet this challenge. **Figure 12** shows the basic code for implementing it.

*Figure 12 Using Dynamic Tokens to Specify Internal Table Sort Order*

```
* dynamic sort
name = 'AGE'.
…
SORT itab BY (name).

* static sort
SORT itab BY age.
```

First, the name of the component to use for sorting the internal table ITAB is stored in the string variable NAME at runtime. According to syntax rules for a dynamic statement, you *must* write the component name in capital letters. Instead of statically specifying the component name in the SORT statement, the string variable containing the component name is specified in parentheses, with no spaces between the variable name and the parentheses.

So you can compare the dynamic statement with its static counterpart, I added the corresponding static statement at the end of the example.

The best way to conceptualize dynamic token specification is to think of it as a replacement. The statement is completed at runtime by replacing the variable in parentheses with its value. In the example shown in Figure 12, (name) is replaced by age.

Clearly, using dynamic token specification is both valuable and powerful. However, be aware of the consequences of being unable to determine the ABAP statement at compile time:

- No static type check or syntax check is performed for the dynamic parts of a statement.
- Runtime errors can occur if the dynamic statement is not valid when completed. However, most runtime errors can be caught. Refer to the SAP application server documentation in the ABAP Developer Workbench for statement-specific guidelines.

ABAP offers five forms of dynamic token specification, each of which is intended for a specific part of a program statement:

1. **Dynamic field specification** contains the name of a field. The field for an ASSIGN statement, which should be assigned to a field symbol, can be specified dynamically.
2. **Dynamic type specification** contains the name of a type. The type for CREATE DATA can be specified dynamically.
3. **Dynamic component specification** contains the name of a component of a structure, such as the sort order for an internal table (see the example in Figure 12).
4. **Dynamic clause specification** contains a whole part of a statement. For example, all clauses of the Open SQL statement SELECT can be specified dynamically. In this case, the variable must be an internal table of character fields.
5. **Dynamic subroutine specification** contains the name of a subroutine. Methods, functions, forms, programs, and transactions can be called dynamically.

To determine whether a statement supports dynamic token specification, refer to the ABAP Workbench Online Help.

Now let's see how you would use dynamic tokens in the real world. In **Figure 13**, dynamic token specification, field symbols, and data references are used to implement a display program for any database table. It also illustrates using a WHERE clause as an input parameter.

This program takes a database table name and a WHERE clause as input parameters. The parameters are initialized with values for the database table SNAPT. (Note that SNAPT is just an example; it contains the static text for all short dumps.)

First, an appropriate work area is created with dynamic type specification using the name of the database table. For every database, a corresponding ABAP structure type with the same name exists. The field symbol <LINE> is used to access the work area. Because the WHERE clause demands a table for dynamic clause specification, the parameter P_WHERE is converted into a table WHERE_TAB. The SELECT loop, which contains the dynamic specification of the database table name and the

*Figure 13 Using Dynamic Tokens, Field Symbols, and References to Implement a Table Display Program*

```
PROGRAM dynamic_select_example.

PARAMETER:
  p_from(30)  TYPE c DEFAULT 'SNAPT',
  p_where(255) TYPE c
    DEFAULT 'LANGU = ''DE'' AND ERRID = ''CREATE_DATA_UNKNOWN_TYPE'' '.

CLASS lcl_util DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      write_struct IMPORTING p_struct TYPE any.
ENDCLASS.

CLASS lcl_util IMPLEMENTATION.
  METHOD write_struct.
    FIELD-SYMBOLS:
      <field> TYPE ANY.
    WRITE / '('.
    DO.
      ASSIGN COMPONENT sy-index OF STRUCTURE p_struct TO <field>.
      IF sy-subrc <> 0.
        EXIT.
      ENDIF.
      WRITE /4 <field>.
    ENDDO.
    WRITE / ')'.
  ENDMETHOD.
ENDCLASS.

DATA:
  data_ref TYPE REF TO data,
  where_tab LIKE TABLE OF p_where.

FIELD-SYMBOLS:
  <line> TYPE ANY.

START-OF-SELECTION.
  CREATE DATA data_ref TYPE (p_from).
  ASSIGN data_ref->* TO <line>.
  APPEND p_where TO where_tab.
  SELECT * FROM (p_from) INTO <line> WHERE (where_tab).
    CALL METHOD lcl_util=>write_struct EXPORTING p_struct = <line>.
  ENDSELECT.
```

Declare input parameters for the database table name (default SNAPT) and a WHERE condition with a default.

Declare the utility class for the WRITE_STRUCT method for outputting structures.

Implement the output method for writing a structure to the ABAP List using field symbols and ASSIGN COMPONENT.

Use dynamic token specification to dynamically create the work area, and SELECT with a dynamic database name and WHERE condition.

`WHERE` clause, retrieves the entries from the database. Every retrieved entry is written to the ABAP List by the method `WRITE_STRUCT` of the class `LCL_UTIL`. (Refer back to the section on field symbols if you need a reminder of how some of this code works.)

There is one more important concept to cover before concluding with how to generate programs dynamically. You now know how to access or operate on data dynamically. Let's see what is required when you need to get information at runtime about data that is passed with generic types.

## Runtime Type Identification (RTTI)

Runtime Type Identification (RTTI) is a powerful technique for obtaining all information about a data type at runtime. When you use dynamic programming techniques, sometimes you need to dynamically determine the data type or properties in order to decide how to handle the data. This situation typically occurs when you use generic types, where you need to obtain at runtime the missing data characteristics not described by the generic type. For example, if you use the generic type ANY in a subroutine, you will need to get information at runtime about the type of the data being passed.

To use RTTI, you need to first understand that a type is described by an object. In turn, that object contains all information about a type in its attributes (name, type, length, etc.). You can derive the description object for a type from either the data or the name of the type. You can also navigate between description objects. For example, a description object for the line type of an internal table type can be derived from the description object of the internal table type, or a description object for the component type of a structure type can be derived from the structure type, and so on.

RTTI is implemented in ABAP Objects. Description objects for types are created from description classes, and every type in the ABAP hierarchy has a corresponding description object. Every description class has special attributes and appropriate navigation methods. For example, the class CL_ABAP_TABLEDESCR has an attribute TABLE_KIND and a navigation method GET_TABLE_LINE_TYPE.

**Figure 14** shows the RTTI class hierarchy. As you can see, the class CL_ABAP_TYPEDESCR is the root class, which contains all methods to derive a description object from a data type or the name of the type.

The class CL_ABAP_TYPEDESCR provides four methods to derive a description object for a type:

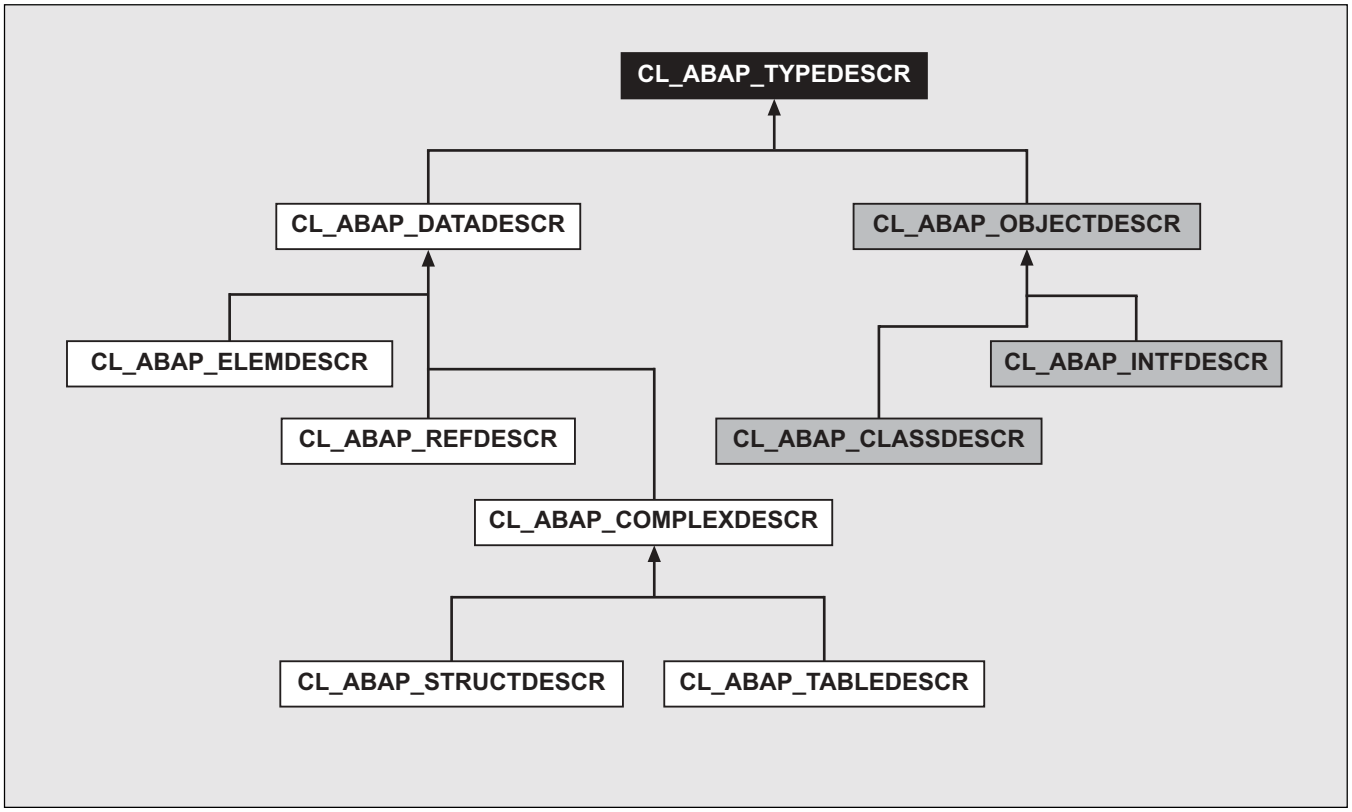
- DESCRIBE_BY_NAME: This method takes a name of a type as an input parameter and returns an object reference to the corresponding description object.
- DESCRIBE_BY_DATA: This method takes data as input and returns an object reference to the description object of the data type.
- DESCRIBE_BY_DATA_REF: This method takes a data reference as input and returns an object reference to the description object of the data type for the data pointed to by the reference.
- DESCRIBE_BY_OBJECT_REF: This method takes an object reference as input and returns an object reference to the description object of the object type for the object pointed to by the reference.

You can obtain a reference to a description object for a type in different ways (e.g., by name, by data, with navigation, etc.). RTTI guarantees that exactly one description object exists for every type. All references point to the same description object, no matter which path was chosen to retrieve it.

The next example shows how to use RTTI to implement a strict parameter check. ABAP performs only a technical parameter check. In other words, types with the same features are considered to be compatible, even if they have different names. If a method relies on a passed parameter being typed with a specific type because of other features, it

*Figure 14 The RTTI Class Hierarchy*




cannot be expressed in ABAP. In the example in **Figure 15**, RTTI is used to implement a strict type check. If the type of the passed parameter is not exactly the same as demanded, it generates a runtime error.

In this example, the class LCL_TYPECHECK implements a strict type check. The CONSTRUCTOR method takes the type name to check for as input. After creating an object of LCL_TYPECHECK and passing the type name, the method CHECK is called.

*Figure 15 Using RTTI to Implement a Strict Type Check*

```
PROGRAM rtti_typecheck_example.

TYPE-POOLS abap.

CLASS lcl_typecheck DEFINITION.
  PUBLIC SECTION.
    METHODS:
      Constructor
        IMPORTING p_type_name TYPE string
        EXCEPTIONS type_not_found,
```

Declare the class LCL_TYPE_CHECK with the CONSTRUCTOR and CHECK methods. The CONSTRUCTOR method initializes the type that is checked by the CHECK method.

*(continued on next page)*

*Figure 15 (continued)*

```
      check
        IMPORTING p_x TYPE any
        RETURNING value(p_flag) TYPE abap_bool.
  PRIVATE SECTION.
    DATA:
      rtti_ref TYPE REF TO cl_abap_typedescr.
ENDCLASS.

CLASS lcl_typecheck IMPLEMENTATION.
  METHOD constructor.
    CALL METHOD cl_abap_typedescr=>describe_by_name
      EXPORTING p_name = p_type_name
      RECEIVING p_descr_ref = rtti_ref
      EXCEPTIONS type_not_found = 4.
    IF sy-subrc <> 0.
      RAISE type_not_found.
    ENDIF.
  ENDMETHOD.
  METHOD check.
    DATA check_ref TYPE REF TO cl_abap_typedescr.
    CALL METHOD cl_abap_typedescr=>describe_by_data
      EXPORTING p_data = p_x
      RECEIVING p_descr_ref = check_ref.
    IF check_ref = rtti_ref.
      p_flag = abap_true.
    ELSE.
      p_flag = abap_false.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

TYPES:
  my_special_type TYPE HASHED TABLE OF i
    WITH UNIQUE KEY table_line.

CLASS lcl_app DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS except_only_my_special_type
      IMPORTING p_x TYPE my_special_type.
ENDCLASS.

CLASS lcl_app IMPLEMENTATION.
  METHOD except_only_my_special_type.
    DATA my_type TYPE REF TO lcl_typecheck.
    CREATE OBJECT my_type EXPORTING p_type_name =
      'MY_SPECIAL_TYPE'.
    IF my_type->check( p_x ) = abap_true.
      WRITE / 'Typecheck OK'.
```

The private attribute RTTI_REF keeps a reference to the type description object.

Implement the class TYPE_CHECK. The CONSTRUCTOR method retrieves the type description object, which is used by the CHECK method. The type description reference is stored in RTTI_REF.

The CHECK method retrieves the type description object of the parameter. The check is done only by comparing this reference with RTTI_REF, because for the same type the same description object is returned.

Declare the application class, which uses the class CHECK to check whether the parameters passed to it are typed with MY_SPECIAL_TYPE. (Note: ABAP does only a technical type check!)

Implement the application method. First create an instance of the CHECK class, then call the CHECK method for parameters. The application only returns OK if the CHECK method returns true.

*Figure 15 (continued)*

```
    ELSE.
      WRITE / 'Typecheck ERROR'.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

DATA:
  my_special_data TYPE my_special_type,
  some_other_data TYPE HASHED TABLE OF I
    WITH UNIQUE KEY table_line.

START-OF-SELECTION.
  CALL METHOD lcl_app=>except_only_my_special_type
    EXPORTING p_x = my_special_data.
  CALL METHOD lcl_app=>except_only_my_special_type
    EXPORTING p_x = some_other_data.
```

Call the application with a parameter of type MY_SPECIAL_TYPE and a parameter of another type that is only technically compatible.

It checks whether the data type of the passed data P_X is the same as the type passed at the creation of the CHECK object.

In the CONSTRUCTOR method, the object reference to the description object of the passed type is stored in the private instance attribute RTTI_REF. The method CHECK gets the description object of the data type. Because RTTI guarantees that only one description object exists for every type, the type check itself consists only of one object reference comparison. If the data passed to the method CHECK is of the same type as the description object pointed to in the attribute RTTI_REF, the method DESCRIBE_BY_DATA returns a reference to the same description object. The CHECK method either returns ABAP_TRUE or ABAP_FALSE, depending on the success of the type check.

The class LCL_APP is a sample application to use the class LCL_TYPECHECK. The method EXCEPT_ONLY_MY_SPECIAL_TYPE takes one input parameter. If the parameter was typed with MY_SPECIAL_TYPE, a success message is written to the ABAP List. Otherwise, an error message is written to the list.

Note that the program calls the example application class twice. The first time, it is called with the variable MY_SPECIAL_DATA, which is typed with MY_SPECIAL_TYPE. The second time, it is called with the variable SOME_OTHER_DATA. While both variables technically have the same type (because it can be passed to a parameter of type MY_SPECIAL_TYPE in ABAP), the second variable was not typed with the type MY_SPECIAL_TYPE. Therefore, the application writes a success message for the first case and an error message for the second case.

You could also use RTTI in either of the two following ways:

- Implement other kinds of parameter checks using the technique in the method CHECK of the class LCL_TYPECHECK. You could compare any attributes of the description object to decide whether the check is true or false. You could even implement the ABAP type check in ABAP.
- Graphically represent any ABAP data structure or tree with a generic service. For example, the ABAP Workbench uses RTTI for its services.

You have now been exposed to all the important concepts needed for dynamic programming in ABAP. We began with techniques for accessing data dynamically, starting with generic types and proceeding to the dynamic data types of internal tables and strings. Next, we explored techniques for executing operations dynamically using various forms of dynamic token specification. And we have just discussed using RTTI for dynamically obtaining parameter characteristics.

But sometimes even all these techniques are not sufficient to solve some problems. As a last resort, you might want to consider dynamic code generation and execution, which I'll now discuss briefly.

## Program Generation

In rare cases, dynamic programming features alone are not powerful enough to solve some application design problems. Suppose most of the information you need to write a particular subroutine or program is available only at runtime. In situations like this, where the data structures are unknown and must be created at runtime, program generation may be your only option. One common scenario is migrating legacy data. In these cases, you might consider writing a utility program that subsequently generates and executes at runtime a program with the necessary logic.

Program generation is considered to be the highest level of dynamic programming because the source code is created at runtime and all ABAP features can be used independently of input parameters. But be aware that this is an expensive and difficult option. The code to generate programs dynamically is very complex and hard to maintain.

Even if you adopt this approach, you still have choices to make. ABAP supports two types of runtime program generation — transient and persistent. In closing, let's review the impacts of this choice and contrast the two implementation methods.

### *Transient Code Generation*

In ABAP, you can generate programs at runtime and execute them. To generate a program, the source code must be created at runtime. The source code is then passed to special ABAP commands that generate and execute the program.

You have two options for generating a program during runtime. The difference between them is the lifetime of the generated program. Transient code generation means that the generated program exists only as long as the internal mode exists. When the internal mode is finished, all transient-generated programs are deleted. Persistent code generation means that the generated program is stored permanently in a database (until you delete it).

Generating code at runtime is typically used for handling dynamic data. It is possible to generate very efficient programs for special data at runtime. However, be aware of the following disadvantages:

- Generating a program at runtime is very time-consuming and memory-intensive.
- Generated programs are hard to debug.
- There are no static checks or scans (such as scanning for the use of critical ABAP commands in a system) for programs that are generated at runtime.
- Generated programs are not fully supported by the ABAP Workbench (e.g., there is no Where-used list available).
- Special services, such as the connection to the Transport Organizer, have to be implemented manually.

*Figure 16 Generating and Executing a Transient Program*

```
PROGRAM subroutine_pool_example.

TYPES:
  source_line(72) TYPE c.

DATA:
  src           TYPE TABLE OF source_line,
  prog_name(30) TYPE c,
  msg(120)      TYPE c,
  line(10)      TYPE c,
  word(10)      TYPE c,
  off(3)        TYPE c.

APPEND 'PROGRAM SUBPOOL.'                TO src.
APPEND 'FORM DYN1.'                      TO src.
APPEND '  WRITE / ''Hello, I am DYN1!''.' TO src.
APPEND 'ENDFORM.'                        TO src.

GENERATE SUBROUTINE POOL src
  NAME prog_name MESSAGE msg LINE line WORD word OFFSET off.

IF sy-subrc <> 0.
  WRITE: / 'Error during generation in line', line,
         / msg, / 'Word:', word, 'at offset', off.
ELSE.
  PERFORM dyn1 IN PROGRAM (prog_name).
ENDIF.
```

Store the source code for the SUBPOOL program with the DYN1 form in the internal table SRC.

Generate the program in the internal table SRC.

If there are no errors during program generation, execute the DYN1 form.

- Program generation can be a potential security problem, because it is not possible to statically check the generated programs.

Based on these caveats, the general rule should be obvious: do *not* generate programs unless you have no other option. One exception is working with Open SQL, as not all of its constructs support dynamic features. For example, subqueries, range tables, and for-all entries cannot be specified dynamically. In cases where such features are needed, dynamic program generation can be useful.

A transient-generated program is called a *subroutine pool*. Subroutine pools cannot be called directly, and you can only call forms in a subroutine pool. In addition, there is a maximum limit of 36 subroutine pools per internal mode. **Figure 16** shows how to generate a subroutine pool and execute a form in it.

The program in Figure 16 generates a subroutine pool with the form `DYN1` and executes the form. First, the source code is created and stored in an internal table with a mandatory line type of 72 characters. Every line in the internal table `SRC` represents one line of source code. The form `DYN1` contains only one command, which writes the text `Hello, I am DYN1!` to the ABAP List. The command `GENERATE SUBROUTINE POOL` generates a subroutine pool from the internal table `SRC` and publishes it for the internal mode. The program name

PROG_NAME is an output parameter. The command GENERATE SUBROUTINE POOL generates a unique name for every generated subroutine pool, which is used to call a special form. If a syntax error occurs while the subroutine pool is being generated, the output parameters MSG, LINE, WORD, and OFFS are set appropriately. The SY-SUBRC is set to not equal 0 if an error was detected during the generation.

Persistent code generation is somewhat more complex than transient code generation. Let's look at that option now.

### *Persistent Code Generation*

Persistent code generation has most of the same advantages and disadvantages as transient code generation. However, you can call persistent programs directly, and there is no limit to the number of generated programs within an internal mode.

Be aware that persistent code generation is even more time-consuming and memory-intensive than transient code generation. It involves selected database operations (such as insert report and update report) to achieve the persistence of the generated program. However, it does have one advantage over transient code generation: the program is available to *all* programs within *all* SAP application servers of an SAP system, not only for the internal mode from which it was generated. Thus you might want to use this method if the program will be used by more than one application server, or if it will be used more than once.

**Figure 17** shows you how a persistent program is generated and executed. Note that the program still exists after the end of the execution of the generating program. You must explicitly delete persistent-generated programs with the ABAP command DELETE REPORT.

First the program source is created and stored in the internal table SRC. The special command INSERT REPORT inserts the program into the program database table of the SAP application server. The command GENERATE REPORT generates a program that already exists in the program database table. You choose the program name, which must be a standard program name as in the ABAP Workbench. Any existing programs are overwritten. The command SUBMIT executes the program, the name of which must be passed as a parameter. The AND RETURN statement causes the starting program to resume after the execution of the submitted program.

In summary, program generation is the most powerful and flexible method of dynamic programming. The downside is that these programs are very expensive to create and difficult to maintain. Program generation should always be your last resort for solving a design problem. Remember: when in doubt, do not generate programs.

## *Enhancements in Releases Later Than 4.6*

Most of you are probably running Release 4.6 of the SAP application server and do not have direct access to a later release. However, many dynamic programming features were significantly enhanced in the next release. **Figure 18** provides an overview of these changes to provide insight into the improvements you can expect.

## *Ready to Get Started*

As you can see, ABAP is a complex and powerful programming language for developing business applications. Programming in ABAP, and specifically dynamic programming in ABAP, is too broad to cover thoroughly in a single article. To provide you with a jump-start to dynamic programming in ABAP, I have collected an overview of all the related

*Figure 17 Generating and Executing a Persistent Program*

```
PROGRAM generate_program_example.

TYPES:
  source_line(72) TYPE c.

DATA:
  src TYPE TABLE OF source_line,
  msg(120)      TYPE c,
  line(10)      TYPE c,
  word(10)      TYPE c,
  off(3)        TYPE c.

CONSTANTS:
   prg_name(30) VALUE 'ZDYNGENTEST1'.

APPEND  'PROGRAM ZDYNGENTEST1.'          TO src.
APPEND  'WRITE / ''Hello, I am dynamically created!''.' TO src.

INSERT REPORT prg_name FROM src.

GENERATE REPORT prg_name
  MESSAGE msg LINE line WORD word OFFSET off.

IF sy-subrc <> 0.
  WRITE: / 'Error during generation in line', line,
         / msg, / 'Word:', word, 'at offset', off.
ELSE.

  SUBMIT (prg_name) AND RETURN.

ENDIF.
```

Store the source code for the ZDYNGENTEST1 program in the internal table SRC.

Store the source code for the program in the database.

Generate the program in the database.

Execute the program and return control to the submitting program.

*Figure 18 Enhancements to Dynamic Programming After Release 4.6*

| Feature | Changes After Release 4.6 |
|---|---|
| Generic types | Introduces the new generic types NUMERIC, CSEQUENCE, XSEQUENCE, CLIKE, and SIMPLE. |
| Dynamic types | Provides special string constants for which trailing spaces are significant (i.e., no shift needed to get a string with one space). Strings can be used in databases and with Open SQL. |
| References | Allows reference typing with any type, not just DATA. Supports the use anywhere (not just in the ASSIGN statement) of the special operator "->*" to access data pointed to by a reference. |

concepts in one place. If you are interested in a more detailed discussion of ABAP, I recommend the book *ABAP Objects — An Introduction* (Addison-Wesley) by Horst Keller and Sascha Krueger. (Until now, this book has been available only in German. It will be published in English by Addison-Wesley in the fourth quarter of 2001.)

I hope that this information and the supporting examples based on my experiences give you the confidence to start using dynamic programming techniques in your own ABAP programs. If you have questions or suggestions, I would be very interested in receiving them.

*Holger Janz is a software developer at SAP AG in the Business Programming Languages Group. Prior to joining SAP in 1997, he studied computer science at the University of Rostock and Constance, where he focused on object-oriented programming languages. As a member of the development team at SAP, Holger's responsibilities include parts of the ABAP Virtual Machine and the ABAP Compiler, and the integration of the JavaScript Virtual Machine into the SAP Web Application Server. His team is also responsible for the integration of Java into the SAP Web Application Server architecture. He can be reached at holger.janz@sap.com.*