This sample chapter walks through the new language features introduced in ABAP 7.4 and above. These new features are divided into the following categories: creating data, string processing, calling functions, conditional logic, internal tables, object-oriented (OO) programming, and search helps (plus one final catch-all category for topics that don’t fit into any of the previous ones). For each new feature, examples are provided.

“New Language Features in ABAP”
Chapter 3

New Language Features in ABAP

“It’s a beautiful thing, the destruction of words. Of course the great wastage is in the verbs and adjectives, but there are hundreds of nouns that can be got rid of as well.”

—George Orwell, 1984

New features, new goodies . . . could this be described as a programmer’s idea of heaven? In the early 80s, when I was still a teenager, my parents bought me a BBC Microcomputer to replace my ZX81. Even then, my primary interest was programming, not playing games, and what I liked most about my new toy was the broad range of commands that were then available to me in the BBC’s version of BASIC (e.g., constructs such as REPEAT UNTIL). Even now, many years later, I feel exactly the same when something new and exciting is added to the ABAP language.

As time has gone by, more commands and constructs have been added to ABAP and (until the advent of SAP BTP, ABAP environment) nothing has been taken away, to ensure backward compatibility—and the rate of change seems to be accelerating violently. A fair number of changes were made as a result of the introduction of SAP NetWeaver 7.02, but this is nothing compared to the cascade of changes that came with version 7.4 and continued in versions 7.5 and beyond. This proves beyond a doubt that ABAP is not a dead language—which was a fear of many people in 2001, when there was talk (from the very top of SAP) about replacing ABAP with Java.

The quote at the beginning of the chapter, from George Orwell’s novel 1984, is about the destruction of words. The character working on the Newspeak language says to the main character, Winston Smith, that the ruling party is hell-bent on destroying existing words, as opposed to creating new ones. The idea that began with ABAP 7.4 is, in some ways, the same: many of the 7.4 changes allow you to achieve the exact same tasks as before, but with half the code or less. For example, in Newspeak, “that was wonderful, fantastic, the best thing ever” becomes “++good.” In ABAP, CONCATENATE this into the_other SEPARATED BY ‘_’ becomes the_other = this && ‘_’ && that.

Of course, it’s not all about destroying words; releases 7.4 and up add a lot of brand-new functionality as well. This chapter will focus on the large number of new features that have been introduced into ABAP in the latest releases. These features are divided into the following categories, based on their general area of functionality: creating data, string processing, calling functions, conditional logic, internal tables, object-oriented (OO)
programming, and search helps (plus one final catch-all category for topics that don’t fit into any of the previous ones).

**ABAP Versions**

In the same way that George Orwell’s characters were trying to get rid of words, SAP is increasing the number—at least when it comes to having multiple names for the latest ABAP version. This started with ABAP 7.53, which was also known as 1809 (as it was released in September 2018).

The idea was that thereafter a new version of ABAP would be released on average every three months (which has been the case)—so, 1812, 1903, 1906 and so on. At time of writing, the latest version is 2108.

You get two types of periodic announcements of new ABAP versions, both with very similar naming conventions:

- **Announcements concerning SAP BTP, ABAP environment (ABAP in the cloud),** which come out quarterly and have names like 2012 (for December 2020)

- **Announcements concerning the ABAP platform for SAP S/4HANA (ABAP on-premise),** which come out annually and which used to be named 1909 after the year and month of release, but now have names like 2020 (meaning it was released at the end of 2020)

Each SAP S/4HANA on-premise ABAP version also has an internal SAP number like 7.53 (1809), or 7.54 (1909), or 7.55 (2020). In SAP presentations, the two numbers are used interchangeably—often on successive slides.

Version 7.53/ABAP 1809 strips away the last vestiges of platform-independence. That version of ABAP only works with an SAP HANA database, which in turn only works with a Linux operating system.

If you are in an SAP BTP, ABAP environment system, to see what version you’re on, in ADT click the project on the left-hand side that represents the cloud system, then at the bottom-right-hand side select the **Properties** tab. You’ll see something like Figure 3.1.

**Figure 3.1** Displaying SAP BTP ABAP Version

In this example, the ABAP release is both 7.85 and 2108 at once (they’re the same thing).

---

**3.1 Declaring and Creating Variables**

Recently, SAP jumped up onto the rooftops and shouted through a megaphone that with ABAP 7.5 there’s a new data type, int8, which handles really big numbers (i.e., numbers that are 18 digits long, plus or minus). I presume that’s what’s meant by big data; I had always wondered. Perhaps you need such big integers to record the number of SAP product name changes per year.

Coming down from the roof, one of the main differences between ABAP and other languages is that in ABAP we’ve been taught to declare all our variables at the start of the program—for example, in **TOP INCLUDE**. Other languages declare them just before they’re used, and such variables tend to be more local in scope—for example, within a loop. Despite the official ABAP programming guidelines, many ABAP programmers have taken to declaring variables just before they’re used for the first time, for the purposes of making the program more readable by humans and hence easier to change.

The good news is that as time goes on, changes in the ABAP language make this unofficial practice more and more acceptable. This section will discuss several features that contribute to this shift.

### 3.1.1 Omitting Data Type Declarations

One reason ABAP 7.4 has made it less important to declare variables at the start of your routine or method is that the need for the majority of data declarations has melted away. The compiler knows what data type it wants (it has to know to be able to perform a syntax check), so why not let it decide the data type of your variable? In the examples ahead, you’ll see how letting the compiler decide the data type—instead of declaring it yourself—can save you several lines of code. For example, here is a piece of code declaring a data type:

```
DATA: monster_instructions TYPE string.
```

```
monster_instructions = 'Jump up and down and howl'.
```

Without the data type declaration, you can simplify this to just the following:

```
DATA(monster_instructions) = 'Jump up and down and howl'.
```

Similarly, the following piece of code also declares a data type:

```
DATA: number_of_monsters TYPE i.
```

```
number_of_monsters = LINES( monster_table )
```

And again, you can simplify this:

```
DATA(number_of_monsters) = LINES( monster_table ).
```
As you can see, this system has the potential to dramatically reduce the number of lines in your programs. You’ll see various examples of such inline declarations throughout this chapter because they have applications in many different areas of ABAP programming.

You Aren’t My Type
If you’re creating a domain in ABAP versions 1909 and up, you’ll see when using (F4) help that there are now 37 different data types available, as opposed to the 31 that were there previously. Six native SAP HANA data types have been added. Practically, that means you can (for example) create a timestamp data element using built-in type UTCLONG as opposed to having to use domain T2NTSTMP.

3.1 Declaring and Creating Variables
3.1.2 Creating Objects Using NEW
In the Java programming language, the NEW command is used to create instances of objects (they say that everything in Java is an object). For example, MonsterFred = NEW ( Monster ) in Java creates an instance called Fred of the class Monster. In the same way that the ABAP runtime environment has been shamelessly stealing features from Eclipse, now the ABAP language is stealing keywords from Java—so you now have a NEW command as well.

Previously, you’d use the following code:
DATA: monster TYPE REF TO zcl_monster.
CREATE OBJECT monster EXPORTING name = 'FRED'.

Now, you’ll be able use just the following line:
DATA(monster2) = NEW zcl_monster( name = 'FRED' ).

As you can see, this halves the number of lines of code you need. More importantly, the debate about whether you declare the variables at the start of a routine (per the official ABAP guidelines) or just before the variable (to aid humans who might be reading the code) is no longer relevant. You have the type of the variable right in your face at the instant the variable is created.

3.1.3 Filling Structures and Internal Tables while Creating Them Using VALUE
No doubt you’re familiar with the following common statement:
DATA: monster_name TYPE string VALUE 'FRED'.

In this statement, you create a variable and give it an initial value, which can then be changed later. In the past, the ability to change initial values has only been available for elementary data types. However, as of 7.4, the VALUE statement has come of age, and you now can define the initial values for structures and internal tables.

Listing 3.1 contains a pre-7.4 example of querying a database. In this example, you create a selection table to be used in a SQL query, but you’re only interested in laboratories in which monsters will be created. But because you can’t use the VALUE statement, you have to fill up one or more work areas and then append them to the selection table.

Listing 3.2 Database Query with VALUE

However, as of 7.4, you can use the VALUE statement and thus achieve the same effect with fewer lines of code (see Listing 3.2).
In general, you hard-code the values of lines of internal tables when building up test data for unit tests, as shown in Listing 3.3.

```abap
mt_test_configuration[] = VALUE m_tt_configuration(
  variable_name = 'Monster Model' count = 1 possible_value = 'BTNK'
) (
  variable_name = 'Monster Model' count = 2 possible_value = 'KLKL'
) (
  variable_name = 'Monster Model' count = 3 possible_value = 'ISDD'
) ( variable_name = 'Evilness' count = 1 possible_value = 'EVIL' )
  ( variable_name = 'Evilness' count = 2 possible_value = 'VERY' ).
```

Listing 3.3 Using VALUE to Fill Multiple Lines of Internal Table

Listing 3.3 demonstrates this concept for an internal table, but you can also use this construct to fill structures, and there are three other important points to note about the `VALUE` keyword in that context:

- If filling an existing structure, as opposed to creating a new one, the data structure is blanked out before being filled.
- If you were formerly filling a structure with 100 lines of code, assigning a value to a structure element one at a time, you had to skip over each line manually in the debugger. If you fill them all at once with a `VALUE` statement, there’s only one line for the debugger to skip over.
- When filling a structure with `VALUE`, you need one set of brackets at the start; when filling a table, you need two sets of brackets because you need each row to be enclosed in brackets.

3.1.4 Filling Internal Tables from Other Tables Using FOR

How well I remember starting to program when I was fourteen, with the good old ZX81. The BASIC language I programmed in then had constructs like `FOR x = 1 TO 10`, which meant you were going to loop 10 times, with the variable `X` increasing by one each time. Well, the `FOR` command has now arrived in the ABAP world; let’s look at what it’s all about.

You read about the `VALUE` statement in the last section; you can use it to fill an internal table, as shown in Listing 3.4.

```abap
DATA(table_of_monsters) = VALUE z4tt_monster_header(
  name = 'JIMMY' monster_number = 1
) ( name = 'ROLF' monster_number = 2 ).
```

Listing 3.4 Fill Internal Table Using Hard Coding

That’s great as an example, but in real life you fill internal tables either from the database or from other internal tables; almost never do you fill them with hard-coded values. Prior to 7.4, you could only fill one internal table from another table if the two tables had identical column structures, and you had to add all the lines of one table to another, as follows:

```abap
APPEND LINES OF green_monsters TO all_monsters.
```

Fortunately, thanks to the `FOR` command introduced in ABAP 7.4, you can now do this in a much more elegant way: the tables can have different columns, and you can limit what’s transferred based upon conditional logic. Using the `VALUE` and `FOR` keywords, this will look like Listing 3.5.

```abap
SELECT *
  FROM z4t_monster_head
  INTO TABLE @DATA(all_monsters).

DATA(neurotic_monsters) = VALUE z4tt_monster_header(
  FOR monster_details IN all_monsters WHERE ( sanity_percentage < 20 )
    ( name = monster_details-name
      monster_number = monster_details-monster_number )
  ).
```

Listing 3.5 Filling Internal Tables from Other Tables

3.1.5 Creating Short-Lived Variables Using LET

ABAP 7.4 also allows you to declare variables with short lifespans via the `LET` statement. These variables only exist while creating a variable. (A constructor expression is a mechanism to apply assorted logic when creating a variable, such as an internal table. This is similar to the sort of logic you would find inside a constructor of an OO class, and `VALUE` is an example of a constructor expression.)

To illustrate the use of `LET` statements, say that you have a table of deadly weapons, and you want to arm some monsters. You’re not particularly concerned about which monster gets which weapon, so long as each monster gets one. Listing 3.6 shows some code that creates some very short-lived local variables. The `DATE` `USER` addition translates an internal SAP date into the format in the user settings (e.g., MM/DD/YYYY).

```abap
SELECT *
  FROM z4t_monster_head
  INTO TABLE @DATA(all_monsters).

DATA(iterator) = NEW lcl_weapon_iterator( ).

DO lines(all_monsters[]) TIMES.
  DATA(arming_description) = CONV string(
    LET weapon_name = iterator-get_next_weapon( )
      monster_name = all_monsters[ sy-index ]-name
      date_string =
```
New Language Features in ABAP

3.1 Declaring and Creating Variables

In the past, you did this by declaring several constants with the possible values, but elsewhere in the code nothing stopped values other than the constants being assigned to the variable.

Other languages, like Java, have had the concept of *enumerations* to solve these sorts of problems for years, and with release 7.51 this concept comes to the ABAP language as well.

Simply put, an enumeration is a place in your code where you list the possible values a semantic concept (like monster brain size) can have. Listing 3.7 shows how to define an enumeration; in essence, it’s an extension of the existing *TYPE* construct.

```abap
TYPES: BEGIN OF ENUM monster_brain_size,
    normal, small, micro,
END OF ENUM monster_brain_size.

DATA: brain_size TYPE monster_brain_size.
```

Listing 3.7 Basic Enumeration

Normally, you declare the _TYPE_ of a variable to be either an elementary type like an integer or a data element (with the underlying type derived from the associated domain). In Listing 3.7, you type the variable based on the enumeration. In this case, the _TYPE_ of the variable will be an integer.

“Why an integer?” I hear you ask. Well, what’s happened under the hood is that three constants have been declared:

- **NORMAL = 0**
- **SMALL = 1**
- **MICRO = 2**

Now if you try something like this:

```abap
brain_size = 4
```

You’ll get a syntax error—or indeed a runtime error (which you can catch with exception class **CX_SY_CONVERSION_NO_ENUM_VALUE**) if the source value is only determined at runtime. Instead, programmers are forced to write something like the following:

```abap
brain_size = normal.
```

This also forces the code to look more like plain English, so you’re improving stability and readability at the same time.

The spooky thing is that if you pass the constant into a text field like a string, it magically transforms from an integer like 0 to the text string **NORMAL**—that is, the name of the constant.
At this point, you might be saying that this example isn’t very realistic. You’re correct: monster brain size isn’t an integer, it’s a 10-character field. Of course it is; that goes without saying.

An enumeration doesn’t have to be an integer. You can define it as a different sort of data type, as in Listing 3.8, in which we wanted a 10-character field but ran into a syntax error as soon as the field length was above eight characters. The reason for this is that, as of yet, enumerations aren’t really like domains; you’re not supposed to be interested in the actual value of the constant an enumeration value represents, just its name.

```abap
TYPES:
  brain_size TYPE c LENGTH 8,
  BEGIN OF ENUM monster_brain_size BASE TYPE brain_size,
    normal VALUE IS INITIAL,
    small VALUE 'SMALL',
    micro VALUE 'MICRO',
  END OF ENUM monster_brain_size.

DATA(brain_size) = NEW monster_brain_size( small ).
```

Listing 3.8 Noninteger Enumeration

You might be asking why NORMAL has no value: it’s because one of the values has to be initial, so usually the default value gets that honor.

Now let’s take this to the next level (of code readability) and define the enumeration as not a set of constants, but a structure of constants, as in Listing 3.9.

```abap
TYPES: BEGIN OF ENUM monster_brain_size
  STRUCTURE brain_sizes,
    normal,
    small,
    micro,
  END OF ENUM monster_brain_size
  STRUCTURE brain_sizes.

DATA(brain_size) = brain_sizes-small.
```

Listing 3.9 Enumeration with Structure

As mentioned, at the moment, domains and enumerations are two totally unrelated things, despite both appearing to do the same job, but in the future we’re likely to get automatic linking with actual DDIC domains.

We’re also promised an even better sort of constant: one that gets its value dynamically assigned at runtime but has an immutable value once created. If and when that happens, I’ll dance around the room with joy.

### 3.2 String Processing

The English comedy trio The Goodies once sang:

> String, string, string, string, everybody loves string,
> String for your pants, string for your vest!
> Everybody knows—string is best!

If you’re anything like me, you may spend half your life calling the `CONVERSION_EXIT_ALPHA_INPUT` and `CONVERSION_EXIT_ALPHA_OUTPUT` functions to add and remove leading zeroes from document numbers such as delivery numbers. For example, you might remove the zeroes when showing messages to the user, but then then add them back before you read the database (see Listing 3.10).

```abap
DATA: monster_number TYPE z4de_monster_number VALUE '0000000001'.

"Remove Leading Zeroes ready for Output to User
CALL FUNCTION 'CONVERSION_EXIT_ALPHA_OUTPUT'
  EXPORTING
    Input = monster_number
  IMPORTING
    output = monster_number.

message_container->add_message_text_only(
```
3 New Language Features in ABAP

3.3 Calling Functions

You often have to jump through hoops to play around with the variables in your programs before you can call other methods or functions. Luckily for you, this only gets easier over time. This section will discuss the new ABAP functionalities that make calling functions much easier than it has been in the past.

3.3.1 Avoiding Type Mismatch Dumps when Calling Functions

It can drive you up the wall when you’ve declared a variable that you want to either pass into or get back from a method of a function module and the variable type doesn’t match the expected parameter type. With a method, you get a syntax error; with a function module, you get a short dump at runtime. To be safe, you have to keep jumping in and out of the function module signature to see how the variables were typed. This process is tedious, so it’s led to people creating custom patterns, in which you enter a function module name, the signature is read, and then a whole list of variables is declared with the same type as all the function parameters. Well, half of the problem has now gone away, because if the sole purpose of a variable is to receive a value from a method or function, then the variable can be declared inline and the type read from the parameter definition.

To see this new functionality, look at Listing 3.12. This is an example of what most people have always done: declare some local variables to be passed into a method, cross your fingers, and hope you’ve declared the variable the same way as the input parameter. If not, you get a syntax error.

```
DATA: monster_number TYPE z4de_monster_number VALUE '0000000001'.
```

When using the method shown in Listing 3.11, the monster number variable is totally unchanged. Only the output to the end user is affected, so you can read the database straightaway if you so desire.

```
Listing 3.10 Removing and Adding Leading Zeroes by Function Call
```

```
Listing 3.11 Removing Leading Zeroes via Formatting Option
```

```
Listing 3.12 Declaring Variables with (Hopefully) Same Types as Method Parameters
```

With ABAP 7.4, however, you can accomplish the same thing by declaring the variables returned from the method not at the start of the routine but rather at the instant their values are filled by the method, as shown in Listing 3.13.

```
DATA: monster_number TYPE z4de_monster_number VALUE '0000000001'.
```

```
Listing 3.13 Declaring Variables inline
```

match the expected parameter type. With a method, you get a syntax error; with a function module, you get a short dump at runtime. To be safe, you have to keep jumping in and out of the function module signature to see how the variables were typed. This process is tedious, so it’s led to people creating custom patterns, in which you enter a function module name, the signature is read, and then a whole list of variables is declared with the same type as all the function parameters. Well, half of the problem has now gone away, because if the sole purpose of a variable is to receive a value from a method or function, then the variable can be declared inline and the type read from the parameter definition.

To see this new functionality, look at Listing 3.12. This is an example of what most people have always done: declare some local variables to be passed into a method, cross your fingers, and hope you’ve declared the variable the same way as the input parameter. If not, you get a syntax error.

```
DATA: monster_number TYPE z4de_monster_number VALUE '0000000001'.
```

When using the method shown in Listing 3.11, the monster number variable is totally unchanged. Only the output to the end user is affected, so you can read the database straightaway if you so desire.

```
Listing 3.10 Removing and Adding Leading Zeroes by Function Call
```

```
Listing 3.11 Removing Leading Zeroes via Formatting Option
```

```
Listing 3.12 Declaring Variables with (Hopefully) Same Types as Method Parameters
```

With ABAP 7.4, however, you can accomplish the same thing by declaring the variables returned from the method not at the start of the routine but rather at the instant their values are filled by the method, as shown in Listing 3.13.

```
DATA: monster_number TYPE z4de_monster_number VALUE '0000000001'.
```

```
Listing 3.13 Declaring Variables inline
```
### 3.3 Calling Functions

#### 3.3.2 Using Constructor Operators to Convert Strings

Often, one routine consists largely of a call to several smaller routines (such as **FORM** routines, function modules, and methods). The problem with this is that sometimes the result of one routine has to have its type converted before it can be passed into another routine (e.g., the period from a standard SAP ERP Financials function tends to be two characters long, but if you want to pass that period into a controlling function, then it needs to be three characters long).

Another even more common example is that often you have a variable that is a string, or maybe **NUMC4**, and you want to pass that variable into a function that only accepts input of a certain type (say, **CHAR60**). You can’t pass the variable directly; you have to move it into a helper variable, as shown in Listing 3.14.

#### 3.3.3 Functions Expecting **TYPE REF TO DATA**

When you import parameters of functions or methods, a specific type of parameter is usually required. But in some situations you don’t know the variable type until runtime. In such cases, the only way to achieve what you want is to use dynamic programming.

In cases in which you don’t know the exact data type until runtime, often the importing parameter of the method is typed as **TYPE REF TO DATA**. This is what happens, for example, when you call methods that build up a dynamic signature to pass into a dynamically defined method. **BOPF**, which you will read about later in the book, uses this mechanism all the time. You also may have used parameters that declared **TYPE REF TO DATA** when you needed to store an arbitrary number of values of different data types in a log along with their descriptions.
In pre-7.4 ABAP, you would do something along the lines of the code in Listing 3.16 to satisfy the requirement of a method that expects to receive a parameter of the TYPE REF TO DATA type.

```abap
DATA: bopf_monster_header_records TYPE z4tt_monster_header,
     header_record_reference TYPE REF TO data.

LOOP AT bopf_monster_header_records INTO DATA(bopf_monster_header_record).
  "Before 7.4
  CREATE DATA header_record_reference LIKE bopf_monster_header_record.
  GET REFERENCE OF bopf_monster_header_record INTO header_record_reference.

  "After 7.4
  header_record_reference = REF #( bopf_monster_header_record ).

  "IS_DATA is type REF TO DATA
  io_modify->update(
    iv_node = is_ctx-node_key
    iv_key = bopf_monster_header_record-key
    is_data = header_record_reference ).

ENDLOOP.
```

**Listing 3.16 Filling TYPE REF TO DATA Parameter**

As you can see in Listing 3.16, now you can get rid of most of the code and still achieve the same thing by using the constructor operator `REF` and the hash (`#`) symbol. Because the runtime system knows the data type of `VALUE`, the `REF#` function can read this and create the data object, which is then passed into the method that expects a TYPE REF TO DATA object to be passed in.

### 3.4 Conditional Logic

By now, it won’t come as a shock that again I’m going to mention the ZX81. The language it used back in 1981 was able to handle statements like `IF ( A + B ) > ( C + D ) THEN...` Later, in 1999, when I started programming in ABAP, I missed this ability. Luckily, with the advent of ABAP 7.02, which I first had access to in 2012, I was once again able to do this sort of thing. (It might have taken 31 years for them to work out how to do this, but it was worth the wait.)

In ABAP 7.40, the `IF/THEN` and `CASE` constructs you know and love keep on getting easier. The next sections will explain how.
That said, you have to be really careful when using this syntax; it only makes sense when the functional method is passing back a parameter typed as ABAP_BOOL. As an example, consider the code in Listing 3.19.

```
DATA(monster) = NEW zcl_monster_model( ).
IF monster->wants_to_blow_up_world( ).
  DATA(massive_atom_bomb) = NEW lcl_atom_bomb( ).
  massive_atom_bomb->explode( ).
ENDIF.
```

Listing 3.19 IF Statement without Proper Check

If the functional method in Listing 3.19 returns a string and that string is NO NO! Do not blow up the world, whatever you do, do not blow up the world, then the result is NOT INITIAL and thus evaluated as true, and so it's curtains for all of us.

### 3.4.2 Using XSDBOOL as a Workaround for BOOLC

Another common situation with respect to Boolean logic (or the lack thereof) within ABAP is a case in which you want to send a TRUE/FALSE value to a method or get such a value back from a functional method. In ABAP, you can't just say something like the following:

```
RF_IS_A_MONSTER = ( STRENGTH > 100 AND SANITY < 20)
```

But in some programming languages, you can do precisely that (can you guess which computer could do that in 1981?). Again, we have a workaround in the form of the built-in BOOLC function (see Listing 3.20).

```
* Do some groovy things
* Get some results
* Postconditions
zcl_dbc=>ensure( that = 'A result table is returned'
which_is_true_if = boolc( rt_result[] IS NOT INITIAL ) ).
```

Listing 3.20 BOOLC

In Listing 3.20, you pass in a TRUE/FALSE value based on whether an internal table has any entries. This works fine.

But what if you want to test for a negative using this method? Say, for example, that you want to pass into the IF_TRUE parameter a TRUE/FALSE value that's true if the table is empty. If you use the previous technique using BOOLC, then things start going horribly wrong. This can be demonstrated by running the code in Listing 3.21.

```
DATA: empty_table TYPE STANDARD TABLE OF ztmonster_header.
IF boolc( empty_table[] IS NOT INITIAL ) = abap_false.
  WRITE:/ 'This table is empty'.
ELSE.
  WRITE:/ 'This table is as full as full can be'.
ENDIF.
```

Listing 3.21 Testing for Negative

When you run this code, the output is as follows:

1. This table is as full as full can be.
2. 1 equals 2, and the world is made of snow.

Oh, dear! The reason for this outcome is a fundamental design flaw in the built-in function BOOLC. Instead of returning a one-character field defined in the same way as ABAP_BOOL, it returns a string. If the string is an X (TRUE), then all is well—but in ABAP, comparing a string of one blank character with the blank character inside ABAP_FALSE means that the comparison fails, even though the values are identical.

Therefore, given that a real Boolean variable is out of the question for whatever reason, a new workaround is needed. Fixing BOOLC so that it returns an ABAP_BOOL value would have been too easy, so in ABAP 7.4 a newly created built-in function was added, called XSDBOOL, which does the same thing as BOOLC but returns an ABAP_BOOL type parameter.

You can see an example of its use in Listing 3.22. The function was not invented for this purpose, but it works, and that's all that matters.

```
* Then do the same using XSDBOOL.
  IF xsdbool( empty_table[] IS NOT INITIAL ) = abap_false.
    WRITE:/ 'This table is empty'.
  ELSE.
    WRITE:/ 'This table is as full as full can be'.
  ENDIF.
  IF boolc( 1 = 2 ) = abap_false.
    WRITE:/ '1 does not equal 2'.
  ELSE.
    WRITE:/ '1 equals 2, and the world is made of snow'.
  ENDIF.
```

Listing 3.22 Using XSDBOOL for Correct Logic Test Results
When you run this code, the output is as follows:

1. This table is empty.
2. 1 does not equal 2.

Even better: if you use the wrong one (BOOLC) in ABAP in Eclipse, you get a warning that tells you to use XSBOOL instead.

### 3.4.3 The SWITCH Statement as a Replacement for CASE

How many times have you seen code like that in Listing 3.23? Here you’re getting one value and using a CASE statement to translate that value. The problem is that you need to keep mentioning what variable you’re filling in every branch of your CASE statement.

* Use adapter pattern to translate human readable CRUD
standard values to the BOPF equivalent

```abap
CASE id_edit_mode.
  WHEN 'R' 'Read'
    bopf_edit_mode = /bobf/if_conf_c=>sc_edit_read_only.
  WHEN 'U' 'Update'
    bopf_edit_mode = /bobf/if_conf_c=>sc_edit_exclusive.
  WHEN OTHERS.
    *Unexpected Situation
    RAISE EXCEPTION TYPE zcx_4_monster_exceptions.
ENDCASE.

Listing 3.23 Filling in Variable Using CASE Statement
```

As mentioned in the code, this is the adapter pattern, very common in OO programming. In 7.4, this can be slightly simplified by using the new SWITCH constructor operator, as shown in Listing 3.24.

* Use adapter pattern to translate human readable CRUD
* standard values to the BOPF equivalent

```abap
DATA(bopf_edit_mode) =
  SWITCH /bobf/conf_edit_mode( id_edit_mode).
  WHEN 'R' THEN /bobf/if_conf_c=>sc_edit_read_only
  WHEN 'U' THEN /bobf/if_conf_c=>sc_edit_exclusive
  ELSE THROW zcx_4_monster_exceptions
ENDSWITCH.

Listing 3.24 Filling in Variable Using SWITCH Statement
```

As you can see from this example, the data definition for BOPF_EDIT_MODE (/bobf/conf_edit_mode in this case) has moved into the body of the expression, thus dramatically reducing the lines of code needed. In addition, Java fans will jump up and down with joy to see that instead of the ABAP term RAISE EXCEPTION TYPE we now have the equivalent Java term, THROW. The usage is identical, however; the compiler evaluates the keywords RAISE EXCEPTION TYPE and THROW as if they were one and the same. As an added bonus, this actually makes more grammatical sense, because THROW and CATCH go together better than RAISE EXCEPTION TYPE and CATCH. (It’s lucky that exception classes have to start with CX; otherwise some witty programmer at SAP would create an exception class called UP.)

It’s important to note that the values in the WHEN statements have to be constants, as in the preceding example. If you put something like MONSTER->HEAD_COUNT after the WHEN statement, then the SWITCH statement as a whole explodes and gives an incorrect error message saying “HEAD_COUNT” is unknown, when what it really means is that MONSTER->HEAD_COUNT isn’t a constant. If you need a WHEN statement with variables, you have to use the COND statement described in the next section.

To move away from monsters for a second, as painful as that is, here’s an example of combing two new ABAP constructs together. Let’s say you wanted to merge some values of different lengths into a uniform format. In standard SAP, table VBPA is a good example. It has customer (KUNNR) values, which are 10 characters long, and personnel number (PERRN) values, which are eight characters long. In Listing 3.25, we fill new table LT_VAKPA with the data from VBPA, except the KUNDE field will always come out as a 10-character field no matter if a customer number or personnel number was in VBPA.

```abap
LOOP AT lt_vbpa ASSIGNING FIELD-SYMBOL(<ls_vbpa>).
  INSERT VALUE #(vbeln='00' && <ls_vbpa>-pernr)
  WHEN 'AG' OR 'WE' OR 'RG' OR 'RE' THEN <ls_vbpa>-kunr
  ELSE '00' && <ls_vbpa>-perrn
  ENDNER
  adrnr = <ls_vbpa>-adrnr
  INTO TABLE gt_vakpa.
ENDLOOP.

Listing 3.25 Merging Two Formats Using SWITCH
```

### 3.4.4 The COND Statement as a Replacement for IF/ELSE

All throughout this book, you’ll be reading about how resistance to change is bad. Despite that, let’s admit it: Change can be annoying, especially when it affects your code. One great example of this fact involves coding a CASE statement based on the assumption that one value is derived from the value of another—before someone
comes along and tells you that the rules have changed and the last value in the CASE statement is only true if it’s a Tuesday. On Wednesday, the value becomes something else.

CASE statements can only evaluate one variable at a time, so in the case of this example you have to change the whole thing into an IF/ELSE construct. That’s not the end of civilization as we know it, but the more changes you have to make, the bigger the risk.

Say that you’re really scared that the logic you’ve been given might change at some point in the future, but nonetheless you start off with CASE, such as in the code in Listing 3.26, which is pre-7.4 code that evaluates the description of a monster’s sanity based on a numeric value.

* Fill the Sanity Description
CAS cs_monster_header-sanity_percentage.
WHEN 5.
  cs_monster_header-sanity_description = 'VERY SANE'.
WHEN 4.
  cs_monster_header-sanity_description = 'SANE'.
WHEN 3.
  cs_monster_header-sanity_description = 'SLIGHTLY MAD'.
WHEN 2.
  cs_monster_header-sanity_description = 'VERY MAD'.
WHEN 1.
  cs_monster_header-sanity_description = 'BONKERS'.
ELSE.
  cs_monster_header-sanity_description = 'RENAME SAP PRODUCTS'.
ENCASE.

LISTING 3.26 CASE Statement to Evaluate Monster Sanity

In 7.4, you can achieve the same thing, but you can do so in a more compact way by using the COND constructor operator. This also means that you don’t have to keep specifying the target variable again and again (see Listing 3.27).

"Fill the Sanity Description"
CAS cs_monster_header-sanity_description =
COND (
  WHEN cs_monster_header-sanity_percentage > 75 THEN 'VERY SANE'
  WHEN cs_monster_header-sanity_percentage > 50 THEN 'SANE'
  WHEN cs_monster_header-sanity_percentage > 25 THEN 'SLIGHTLY MAD'
  WHEN cs_monster_header-sanity_percentage > 12 THEN 'VERY MAD'
  WHEN cs_monster_header-sanity_percentage > 1 THEN 'BONKERS'
  ELSE 'RENAME SAP PRODUCTS').

LISTING 3.27 Using COND Constructor Operator

That looks just like a CASE statement, and the only benefit of the change at this point is that it’s a bit more compact. However, when the business decides that you need to take the day into account when saying if a monster is bonkers or not, you can just change part of the COND construct. In the pre-7.4 situation, you had to give up on the whole idea of a CASE statement and rewrite everything as an IF/ELSE construct. The only change needed to the COND logic is shown in Listing 3.28.

DATA: day TYPE char10 VALUE 'Tuesday'.
* Fill the Sanity Description
  cs_monster_header-sanity_description =
  COND text30(  
    WHEN cs_monster_header-sanity_percentage = 5 THEN 'VERY SANE'  
    WHEN cs_monster_header-sanity_percentage = 4 THEN 'SANE'  
    WHEN cs_monster_header-sanity_percentage = 3 THEN 'SLIGHTLY MAD'  
    WHEN cs_monster_header-sanity_percentage = 2 THEN 'VERY MAD'  
    WHEN cs_monster_header-sanity_percentage = 1 AND  
      day = 'Tuesday' THEN 'HAVING AN OFF DAY'  
    WHEN cs_monster_header-sanity_percentage = 1 THEN 'BONKERS'  
    ELSE 'RENAME SAP PRODUCTS').

LISTING 3.28 COND Constructor Operator with Updated Logic

3.5 Internal Tables

Internal tables are the bread and butter of programming in ABAP. A lot of other languages, like Java, can only dream of having such a thing; they have to deal with various sorts of arrays and stacks and hash browns and what have you. In this section, you’ll learn about some new ABAP functionalities that relate to internal tables.

3.5.1 Table Work Areas

Some time ago, SAP decreed that header lines in internal tables were the work of the devil. This was because the use of header lines led to the existence of two data objects in your program with the exact same name, and this was viewed as confusing. For example, it wouldn’t be uncommon to find a program in which you had an ITAB variable that referred to the internal table as a whole and a work area called ITAB that referred to the current line of the table being processed. Clearly, that’s wrong from an academic point of view—but in real life, ABAP programmers were so used to the idea of the header line that it was a hard habit to shake. This was especially true because you had to explicitly declare a variable to act as the header line. For example:

DATA: monster_record LIKE LINE OF monster_table.
The good news is that now you can avoid having to make that extra variable declaration and still have two differently named variables, one for the table and one for the work area. Listing 3.29 shows the syntax for reading into a work area and looping through a table in release 7.4.

```
DATA: table_of_monsters
TYPE STANDARD TABLE OF z4t_monster_head.

READ TABLE table_of_monsters
WITH KEY color = 'RED'
INTO DATA(red_monster_details).

LOOP AT table_of_monsters INTO DATA(loopy_monster_details).
ENDLOOP.
```

**Listing 3.29 Reading into Work Area and Looping through Table**

In Section 3.1.1, you learned that from 7.4 onward you no longer need to perform a `DATA` declaration for elementary data types. It’s exactly the same for the work areas, which are of course structures. If you’re looping into a work area called `MONSTER_RECORD` and the work area structure doesn’t match the type of table `TABLE_OF_MONSTERS`, you’ll see a syntax error.

Therefore, the compiler knows the type it wants—so why do you have to tell it that type in an explicit data declaration? The answer is that now you don’t, and this change will remove all those extra data declaration lines you had to insert when everyone moved away from header lines (if they ever did) and had to explicitly declare work areas instead.

It’s always faster to loop into `FIELD-SYMBOL` rather than a work area even if you don’t want to change any values. Happily, in the same way that you no longer need `DATA` declarations for table work areas, you also no longer need `FIELD-SYMBOL` declarations for the (common) situations in which you want to change the data in the work area while looping through an internal table. In release 7.4, if you want to use field symbols for the work area, then the syntax is as shown in Listing 3.30.

```
READ TABLE table_of_monsters
WITH KEY color = 'RED'
ASSIGNING FIELD-SYMBOL(<red_monster_details>).

LOOP AT table_of_monsters ASSIGNING FIELD-SYMBOL(<loopy_monster_details>).
ENDLOOP.
```

**Listing 3.30 Field Symbols for Work Area**

One vitally important point to note is that if you declare a work area and then loop through an internal area, after the loop is over the work area will be filled with the value of the last row of the internal table that was processed. But if you loop through an internal table into a work area created via the `DATA` statement, then you might wonder if the work area ceases to exist after you leave the loop.

From an academic point of view, not being able to access the work area outside of the loop is a Good Thing, because the textbooks all say that variables should not exist outside their scope (the loop, in this case). Indeed, at SAP, this behavior is very much desired. Unfortunately, the fact is that the variable does exist after you leave the loop. SAP has got around this problem by publishing guidelines saying that if you access such a variable outside of its scope, then you are very naughty, and the staff at SAP will say “tut, tut” if they find out. (Personally, I’m not convinced that this dire threat will have much effect, but I thought I’d let you know.)

### 3.5.2 Reading from a Table

If I told you that you would never have to use the `READ TABLE` statement again to get a line out of an internal table, then perhaps the bottom would drop out of your world. In fact, that’s exactly what happens in ABAP 7.4. You can remove the keywords altogether and replace the English phrase `READ TABLE` with a more computer-like pair of square brackets `[]`.

Prior to ABAP 7.4, calling a line from an internal table would look like Listing 3.31.

```
DATA: monster_name TYPE z4de_monster_name VALUE 'JIMMY',
  table_of_monsters TYPE STANDARD TABLE OF z4t_monster_head,
  monster_details LIKE LINE OF table_of_monsters,
  monster TYPE REF TO zif_4_monster_model.

READ TABLE table_of_monsters INTO monster_details
WITH KEY name = monster_name.

  monster = zcl_4_monster_model->get_instance( monster_details-monster_number ).
```

**Listing 3.31 Reading Line from Internal Table before 7.4**

However, in ABAP 7.4 and up, this is simplified, as shown in Listing 3.32.

```
DATA(monster) = zcl_4_monster_model->get_instance( table_of_monsters[ monster_name ]-monster_number ).
```

**Listing 3.32 Reading Line from Internal Table after 7.40**

**Note**

Some would say that making code less like English and more like machine code actually prevents clarity. For example, if after reading your internal table line you wanted to...
add it to a string and pass the string to a method, then you could go overboard using punctuation marks in your code. The part where you read the internal table gets buried in a morass of code, and someone reading the code might struggle to work out what’s happening:

```abap
ccl_bc_output( |{ monster_name } |"s Monster Number is |{ monster_table [ monster_name ]-monster_number || }).
```

Many people have said the same thing about regular expressions: they get the job done in a very small amount of code, but no one can understand that code.

In real life, the most common usage of the new way of reading internal tables is to read the first line of such a table as follows:

```abap
DATA(selected_monster_line) = selected_monster_lines[ 1 ].
```

What happens if such an expression can’t find the internal table line you’re looking for? The answer is not good news. An exception is raised, whereas you were probably expecting a blank (initial) value to be returned. SAP must have received a lot of complaints about this, because as of 7.4 (SP 8) SAP delivered a workaround: if you add the word OPTIONAL at the end of your query, then suddenly the system doesn’t care if no record is found, and an initial value of whatever data type you’re looking for is returned. This is exactly what would happen with a traditional READ TABLE xyz INTO work_area_abc setup when the read failed and the work area had not yet been filled. An example of this workaround is shown in Listing 3.33.

```abap
DATA(message1) = |{ monster_name }|"s Monster Number is |{ table_of_monsters[ monster_name ]-monster_number OPTIONAL } |

DATA(message2) = |{ monster_name }|"s Monster Number is |{ table_of_monsters[ monster_name ]-monster_number DEFAULT |"9999999999" } |

Listing 3.33 Reading Internal Table with OPTIONAL/DEFAULT
```

The addition of DEFAULT fills in a hard-coded value when the read fails. Then you can check if the read failed by comparing the result with that hard-coded value in the same way you would if SY-SUBRC was 4 when checking if a traditional read on an internal table succeeded.

### 3.5.3 CORRESPONDING for Normal Internal Tables

You no doubt use the ABAP keyword MOVE-CORRESPONDING a lot, moving variable values from one structure to another.

New in 7.4 is a constructor operator called CORRESPONDING without the word move in front. This operator takes moving data between two internal tables to a whole new level. Say that you have two internal tables full of monster-related data, but they’re defined with a different set of columns. What you want to do is copy over the fields with the same names from one table to the other, with two exceptions:

- You don’t want to copy the EVILNESS value from one table to the other, even though both tables have an EVILNESS column.
- You want to copy the column named MOST_PEASANTS_SCARED from one table into a similar column called PEOPLE_SCARED in the second table.

Prior to 7.4, you would declare a bunch of helper variables to store the work areas of the two tables and then loop through the first table, moving everything from one table to another. You’d then perform some logic to deal with your exceptions (see Listing 3.34).

```abap
DATA: green_monsters TYPE STANDARD TABLE OF z4t_monster_head,
     blue_monsters TYPE STANDARD TABLE OF z4t_monster_head.

FIELD-SYMBOLS:
<green_monsters> LIKE LINE OF green_monsters,
<blue_monsters> LIKE LINE OF blue_monsters.

LOOP AT green_monsters ASSIGNING <green_monsters>.
    APPEND INITIAL LINE OF <green_monsters> ASSIGNING <blue_monsters>.
    MOVE-CORRESPONDING <green_monsters> TO <blue_monsters>.
    CLEAR <blue_monsters>-evilness.
    <blue_monsters>-EVILNESS = <green_monsters>-EVILNESS.
    <blue_monsters>-MOST_PEASANTS_SCARED = <green_monsters>-MOST_PEASANTS_SCARED.
ENDLOOP.
```

Listing 3.34 Moving One Table to Another before 7.4

In release 7.4, you can use the constructor operator CORRESPONDING to do the exact same thing, but this time you don’t have to declare the field symbols. You tell the CORRESPONDING operator what the rules are, and it loops through the table for you while executing those rules. Moreover, you don’t even need to define the target table; if you put it after the CORRESPONDING operator, then it creates the target table with the same columns as
the source table, but without the **EVILNESS** column and with the **SCARED** column having a different name but the same type as the source column, as shown in Listing 3.35.

\[
green_monsters = \text{CORRESPONDING #}(
blue_monsters
\text{MAPPING people_scared = most_peasants_scared}
\text{EXCEPT evilness })
\]

Listing 3.35 Moving One Table to Another in 7.4

A common problem is that when you try to do use \text{MOVE-CORRESPONDING} into a hashed or sorted table, then you get a dump because you can’t override the key fields. Using \text{CORRESPONDING} and \text{EXCEPT}, you can prevent this problem—as shown in Listing 3.36, in which you want to either add a new entry to an internal table or update an existing one.

```
"HASHED table
READ TABLE monster_order_items ASSIGNING FIELD-SYMBOL(<order_item>)
WITH TABLE KEY order_number = id_order_number
order_item = id_item_number.

IF sy-subrc = 0.
"Cannot over-write key fields of a Hashed Table
<order_item> = \text{CORRESPONDING #} ( BASE ( <order_item> ) changed_item
EXCEPT order_number order_item ).
ELSE.
INSERT changed_item INTO TABLE monster_order_items.
ENDIF.
```

Listing 3.36 Moving Corresponding Fields to Hashed Table

3.5.4 \textbf{MOVE-CORRESPONDING} for Internal Tables with Deep Structures

In ABAP, a deep structure is not a structure that thinks a lot and has a complex personality but rather is a structure that—in addition to elementary data types, such as strings and numbers—has internal tables. Listing 3.37 shows an example of a deep structure. `T_RESULTS` is defined as a table type, so you have an internal table in which each row has a column that is itself an internal table.

```plaintext
TYPES: BEGIN OF l_typ_monsters,
    monster_number TYPE z4de_monster_number,
    monster_name TYPE z4de_monster_name,
    t_items TYPE z4tt_monster_items,
END OF l_typ_monsters.
DATA: monster_table TYPE STANDARD TABLE OF l_typ_monsters.
```

Listing 3.37 Deep Structure

SAP clearly has been thinking about what happens when you perform a \text{MOVE-CORRESPONDING} maneuver between two slightly different structures that are both deep but defined slightly differently. As you know, \text{MOVE-CORRESPONDING} compares the field names of two structures, and if a field called \text{RESULT} is a string in one and a number in another and the value is \text{XYZ} in the first structure, then trouble looms (i.e., a short dump). Take this one step further: Say you have two internal tables in two deep structures with the same name but defined differently. What’s going to happen when you perform \text{MOVE-CORRESPONDING}? The next examples examine what happens in release 7.02 when moving from one structure to another and then expand to look at how the same process behaves in 7.4. \textit{Spoiler:} In 7.4, you can perform \text{MOVE-CORRESPONDING} between internal tables as a whole, which wasn’t possible in lower versions.

Say that in Europe it’s important to keep track of something called an \textit{IBAN} code for each monster, which is meaningless anywhere else in the world, and in the United States it’s important keep track of a \textit{LOCKBOX} code for each monster, which likewise is meaningless anywhere else in the world. Listing 3.38 sets up structures to store data for each region and then tries to use \text{MOVE-CORRESPONDING} to move some European monster data to the US equivalent.

```plaintext
TYPES: BEGIN OF l_typ_european_monsters,
    monster_name TYPE string,
    monster_iban_code TYPE string,
END OF l_typ_european_monsters.

DATA: iban_code_record TYPE l_typ_european_monsters.

TYPES: BEGIN OF l_typ_european_results,
    laboratory TYPE string,
    t_result TYPE l_tt_european_monsters,
END OF l_typ_european_results.

DATA: european_result TYPE l_typ_european_results,
    european_results TYPE l_tt_european_results.

TYPES: BEGIN OF l_typ_us_monsters,
    monster_name TYPE string,
END OF l_typ_us_monsters.

DATA: lockbox_code_record TYPE l_typ_us_monsters.

TYPES: BEGIN OF l_typ_us_results,
    laboratory TYPE string,
    t_result TYPE l_tt_us_monsters,
END OF l_typ_us_results.

DATA: us_result TYPE l_typ_us_results.
```

Listing 3.38 Moving Corresponding Fields Between Tables
New Language Features in ABAP

```
monster_lockbox_code TYPE string,
END OF l_typ_us_monsters.
```

```
TYPES: BEGIN OF l_typ_us_results,
  laboratory TYPE string,
  t_result TYPE l_tt_us_monsters,
END OF l_typ_us_results.
```

```
TYPES: l_tt_us_monsters TYPE HASHED TABLE OF l_typ_us_monsters
  WITH UNIQUE KEY monster_name.
```

```
TYPES: BEGIN OF l_typ_us_results,
  laboratory TYPE string,
  t_result TYPE l_tt_us_monsters,
END OF l_typ_us_results.
```

```
DATA: us_result TYPE l_typ_us_results,
  us_results TYPE l_tt_us_results.
```

```
european_result-laboratory = 'SECRET LABORATORY 51'.
iban_code_record-monster_name = 'FRED'.
iban_code_record-monster_iban_code = 'AL472111009000000235698741'.
INSERT iban_code_record INTO TABLE european_result-t_result.
MOVE-CORRESPONDING european_result TO us_result.
```

Listing 3.38 MOVE-CORRESPONDING Attempt

What do you think happens? In the US monster structure, the results table has the LOCKBOX field filled with the IBAN code from the European equivalent because there’s a T_RESULT field in both the European and US structures.

Usually, this isn’t what you want. For identical tables, you can always get around this using `TABLE_ONE[] = TABLE_TWO[]`, which changes the first table into an identical copy of the second table. However, this doesn’t work in cases in which, say, you have a database table with 10 fields and an internal table with those 10 fields plus five more fields with text descriptions or calculated fields. Instead, you need to read the database table into one internal table with just the 10 fields and loop through this first internal table, moving the corresponding elements to some sort of second output table, where you fill in extra data, such as names for sales offices and the like and calculated fields.

Thus, instead of the code shown in Listing 3.39, you’d use the code shown in Listing 3.40.

```
MOVE-CORRESPONDING european_results TO us_results.
```

Listing 3.40 Copying between Internal Tables with Different Structures in 7.4

This saves you a few lines of code and is wonderful for when the line structures of the two tables are flat. However, it still doesn’t solve the problem for deep structures. As you’ve seen, MOVE-CORRESPONDING on its own will dump the contents of an internal table component, such as T_RESULT, into the identically named component in the target structure. If T_RESULT in the target has differently named columns, then all sorts of bizarre results might ensue. To counteract this, SAP has come up with two new additions to MOVE-CORRESPONDING in release 7.4 (plus a combination of the two), as follows:

- **MOVE-CORRESPONDING EXPANDING NESTED TABLES**
  
  In MOVE-CORRESPONDING EXPANDING NESTED TABLES, anything that might happen to be in the target internal table already is deleted. Columns with simple values, like numbers, are copied to their identically named friends. But when it comes to complex columns such as T_RESULT, only fields that have the same column name inside the nested tables are copied. For example, MONSTER_NAME is copied because there’s an equivalently named column in the target, but MONSTER_IBAN_CODE is not because there’s no column with the same name, only the LOCKBOX.

- **MOVE-CORRESPONDING KEEPING TARGET LINES**
  
  What happens in MOVE-CORRESPONDING KEEPING TARGET LINES is quite strange; if there’s anything already in the target internal table, then it stays there. Then, at the end of the target table, extra rows are added—which is what would happen if you just performed MOVE-CORRESPONDING between the two tables. This is rather like APPEND LINES OF table1 TO table2, except that the two tables have different structures and only identically named components come across.

- **MOVE-CORRESPONDING EXPANDING NESTED TABLES KEEPING TARGET LINES**
  
  As might be imagined, MOVE-CORRESPONDING EXPANDING NESTED TABLES KEEPING TARGET LINES performs both of the previous tasks at once. It acts like APPEND LINES OF table1 TO table2, copying only identically named components to the new rows, but it’s also a bit clever with any internal table components and only copies identically named components within these structures.

3.5.5 Dynamic MOVE-CORRESPONDING

SAP can be like a dragon with two heads, each head telling you the opposite of the other. In 7.4, the guidance was to not use MOVE-CORRESPONDING at all, but you were given new tools with which to do so. In 7.5, this dichotomy becomes more extreme; on the one hand, you’re advised not to use dynamic programming (because it’s not allowed in some SAP S/4HANA scenarios), and yet a new, dynamic MOVE-CORRESPONDING class is given to you.
In the examples in Section 3.5.3, you could hard-code the rules for mapping fields with different names to be copied to one another—like the 12,000 different names SAP gives to SY-MSGNO in different structures.

Now imagine that such a mapping from a really big structure to a smaller one can vary depending on the conditions at runtime. That might seem a rather esoteric requirement, but you won’t have to think too hard to find a real-life example; after all, the monster examples are actually real problems people have to deal with, with the names changed to protect the innocent.

In the next example, as always, the task is building a monster. The configuration table has about 24 different sorts of criminal brains you could use to fill the inside of the monster’s head. You want to narrow this down to the four best selections, defined at runtime, which are passed into a smaller structure for further processing.

Both structures start with the monster number and name. The larger structure has a list of fields describing the serial numbers of the various brains available, and the smaller structure has fields in the format BEST_BRAIN_XX.

You want to move the identically named fields from the big structure to the small structure and then determine dynamically which of the possible brains from the configuration structure to move to the smaller structure. You’ll achieve this by filling a mapping table prescribing which values to copy between differently named fields, calling two methods from CL_ABAP_CORRESPONDING to create a mapping object, and then executing the mapping itself, as shown in Listing 3.41.

### Listing 3.41 Dynamic MOVE-CORRESPONDING Usage

```plaintext
DATA: mapping_record TYPE cl_abap_corresponding=>mapping_info,
  mapping_table TYPE cl_abap_corresponding=>mapping_table.

mapping_record-level = 0.
mapping_record-kind = cl_abap_corresponding=>mapping_component.

mapping_record-dstname = 'BEST_BRAIN_01'.

IF is_customer_requirements-brain_size_desired = 'NORM'.
  mapping_record-srcname = 'BIGGEST_BRAIN'.
ELSE.
  mapping_record-srcname = 'SMALLEST_BRAIN'.
ENDIF.

APPEND mapping_record TO mapping_table.

TRY.
  DATA(dynamic_mapper) = cl_abap_corresponding=>create(
    source = is_possible_brains
    destination = rs_best_brains
    mapping = mapping_table ).
  
  dynamic_mapper->execute(
    EXPORTING source = is_possible_brains
    CHANGING destination = rs_best_brains ).

  CATCH cx_corr_dyn_error ##NO_HANDLER.
    "In real life you need one 
    "Raise a Fatal Error Message
    RETURN.
  ENDTRY.
```

You can also use the EXCEPT option as an input to the KIND parameter in a mapping record to stop certain identically named fields being moved; the LEVEL parameter is used when mapping deep structures. In the preceding example, you could have achieved the same thing using direct assignment with fewer lines of code, but this approach only becomes advantageous when you have really complicated structures with loads of fields. Traditionally, this sort of task has been handled using field symbols; I imagine SAP feels that using this new class makes the code easier to read.

### 3.5.6 New Functions for Common Internal Table Tasks

When working in ABAP code, there are cases in which you want to know exactly in what line of an internal table the data you’re interested in lives. To find this information in pre-7.4 ABAP, you would have written code that declared a helper variable to store the row number of the target record, read the table for no other purpose than to find that row number, and then transferred the system variable that contained the result of the table read into your helper variable, as shown in Listing 3.42.

### Listing 3.42 Finding the Row of an Internal Table

```plaintext
DATA: start_row TYPE sy-tabix,
  table_of_monsters TYPE STANDARD TABLE OF z4t_monster_head,
```
monster_number  TYPE z4de_monster_number VALUE '0000000001'.

READ TABLE table_of_monsters
WITH KEY monster_number = monster_number
TRANSPORTING NO FIELDS.

IF sy-subrc = 0.
  start_row = sy-tabix.
ENDIF.

Listing 3.42 Reading Internal Table to Get Row Number in 7.02
You see the logic in Listing 3.42 a lot when processing nested loops on standard tables—you follow such code with LOOP AT table_of_monsters FROM start_row—but also in a myriad of other use cases. In any event, in release 7.4 this can be simplified by using the built-in function LINE_INDEX, which does the exact same task but without the need to look at the values of SY-SUBRC and SY-TABIX. This new approach is shown in Listing 3.43.

DATA (start_row) =
  line_index( table_of_monsters[ monster_number = monster_number ] ).

LOOP AT table_of_monsters FROM start_row
  ASSIGNING FIELD-SYMBOL(<monster_details>).
  "Do Something"
ENDLOOP.

Listing 3.43 LINE_INDEX
Throughout this chapter, one aim has been to get rid of helper variables. Because LINE_INDEX is a built-in function, it can be used at operand positions, thus negating the need for the helper variable START_ROW. When looping at an internal table from the row formerly stored in START_ROW instead, use the code shown in Listing 3.44.

LOOP AT table_of_monsters
  FROM line_index( table_of_monsters[ monster_number = monster_number ] )
  ASSIGNING FIELD-SYMBOL(<monster_details>).
  "Do Something"
ENDLOOP.

Listing 3.44 Built-In Function LINE_INDEX at Operand Position
This new built-in function also has a friend: LINE_EXISTS. Say you want to see if an internal table already has an entry for the monster at hand. If it does, then you want to modify the existing entry; if not, then you want to add a new entry. The way this was done prior to 7.4 was to first read the internal table to see if there was an existing entry for the monster. If there was no existing entry, then SY-SUBRC would not be zero. You’d react to that by adding a new entry to the table for your monster, as shown in Listing 3.45.

READ TABLE table_of_monsters ASSIGNING <monster_details>
WITH KEY monster_number = monster_number.

IF sy-subrc NE 0.
  APPEND INITIAL LINE TO table_of_monsters
  ASSIGNING <monster_details>.
ENDIF.

ADD 1 TO <monster_details>-sanity_percentage.

Listing 3.45 Checking If Internal Table Line Exists before 7.4
By adding the new built-in function LINE_EXISTS, the code can be changed as shown in Listing 3.46.

IF line_exists( table_of_monsters[ monster_number = monster_number ] ).
  READ TABLE table_of_monsters ASSIGNING FIELD-SYMBOL(<monster_details>)
  WITH KEY monster_number = monster_number.
ELSE.
  APPEND INITIAL LINE TO table_of_monsters
  ASSIGNING <monster_details>.
ENDIF.

Listing 3.46 Checking If Internal Table Line Exists in 7.4
"Why is that better?" I hear you cry. For one thing, it makes it more obvious what you’re trying to achieve. In addition, using LINE_EXISTS instead of SY-SUBRC is more reliable; SY-SUBRC can be dodgy because you never know when someone is going to have the bright idea of inserting some code between your READ statement and the IF statement that evaluates SY-SUBRC. If you use LINE_EXISTS to have the table read and the evaluation all bundled up in the one statement, as in Listing 3.47, then this means that no one can break the statement with a well-intentioned change.

IF line_exists( table_of_monsters[ monster_number = monster_number ] ).
  "Do Something"
ENDIF.

Listing 3.47 Code All in One Line, with No Reliance on SY-SUBRC

3.5.7 Internal Table Queries with REDUCE
In Section 3.1.4, you learned a new way to create internal tables by using a FOR loop. After you’ve filled up your internal tables, though, you often want to query them. Say, for example, that you want to know how many really mad monsters you have. As of 7.4,
you can do this by using a constructor operator called REDUCE, which contains logic to process an internal table and return a single result.

### Functional Programming

The REDUCE statement is a watered-down copy of the identically named statement in JavaScript, which is a language that uses functional programming. ABAP uses imperative programming, so these sorts of functional constructs are a brand-new, foreign invasion into ABAP.

In the example in Listing 3.48, you’ll first say what variable you want created and what type that variable is going to be, then set an initial value for your result variable, and finally loop over the table, performing assorted logic. That sounds really complicated, but, as you’ll see, the code is not.

```abap
*-----------------------------------------------*
* Fill up NEUROTIC_MONSTERS table, then.    *
*-----------------------------------------------*
DATA: neurotic_monsters TYPE STANDARD TABLE OF z4t_monster_head.

monster_no_table 1 = VALUE #( FOR m_no = 1 UNTIL m_no = 11 ( m_no ) ).
monster_no_table 2 = VALUE #( FOR m_no = 1 THEN m_no + 1 UNTIL m_no = 11 ( m_no ) ).
monster_no_table 3 = VALUE #( FOR m_no = 1 WHILE m_no < 11 ( m_no ) ).
```

Listing 3.48 How Many Really Mad Monsters?

The SY-TABIX element after the REDUCE statement defines the type of the variable being created, MAD_MONSTER_COUNT. It also defines the type of the temporary variable after the INIT statement, which is going to be the result; both MAD_MONSTER_COUNT and RESULT have the type SY-TABIX. You then loop over your internal table NEUROTIC_MONSTERS into a dynamically created work area called MONSTER_DETAILS, and every time your IS_1_IT_MAD method comes back with a value of 1 for the monster being processed, you increase the count of the RESULT variable. After the FOR loop is finished, the value of RESULT is copied into MAD_MONSTER_COUNT.

A common use case is where you have a table of (say) 10 monster numbers, and you have to pass that out to an external system as a comma-delimited string. You reduce the table into one big string with the monster numbers separated by commas.

Two negative aspects of the REDUCE statement are that (a) it can be really difficult to understand what’s going on and (b), even more importantly, you can’t debug the FOR/NEXT loop inside it; the entire statement is processed in one hit.

### FOR/NEXT Variations and Their Gotchas

In both VALUE statements and REDUCE statements you can have FOR/NEXT loops, and they come in three flavors:

- IN for looping over internal tables, as you just saw.
- WHILE a logical expression is true, which is rather like the WHILE/ENDWHILE in traditional ABAP. If the logical expression is never true, there will be no loops.
- UNTIL a logical expression is true. There is no equivalent of this in traditional ABAP, though this construct exists in many other languages. There will be always at least one loop, and if the logical condition is never true, there will be an endless loop, just as in a DO/ENDO loop with no exit mechanism, so you have to watch out for that.

The requirement is this: “I want to fill a table of monsters numbers. The first row will have the value 1, and each subsequent row will have a value one higher than the last. When the value gets to 11, do not add that value to the table and stop processing,” In a FOR/NEXT loop, that comes out as follows:

```abap
monster_no_table = VALUE #( FOR m_no = 1 THEN m_no + 1 UNTIL m_no = 11 ( m_no ) ).
```

That is quite confusing for the reader. The assignment directly after the FOR statement sets the value of the first row. The assignment after the THEN statement says we are going to start looping, adding 1 to the number each time. The assignment after the UNTIL statement says we are going to stop once the value reaches 11. The final ( m_no ) at the end is saying add the current value of the variable to the table during each loop pass.

Just to confuse matters even more, the UNTIL variant automatically increases the loop counter whether you want to or not. As an example, the following two statements are functionally identical. They both fill an internal table with the numbers 1 to 10. In effect, if the THEN statement is missing, then it gets magically added at runtime:

```abap
monster_no_table1 = VALUE #( FOR m_no = 1 UNTIL m_no = 11 ( m_no ) ).
monster_no_table2 = VALUE #( FOR m_no = 1 THEN m_no + 1 UNTIL m_no = 11 ( m_no ) ).
```

The WHILE variation does not have such an automatic “hidden” increment, so the following two statements are not functionally identical. The first does nothing at all (there is no THEN statement, so there is no loop), and the second fills an internal table with the numbers 1 to 10:

```abap
monster_no_table 3 = VALUE #( FOR m_no = 1 WHILE m_no < 11 ( m_no ) ).
monster_no_table 4 = VALUE #( FOR m_no = 1 THEN m_no + 1 WHILE m_no < 11 ( m_no ) ).
```

### 3.5.8 Grouping Internal Tables

You’re most likely familiar with the GROUP BY addition you can add to a database SELECT statement, which condenses similar records into a single row. In the past, when you
wanted something similar to an internal table, you’d use the COLLECT statement or the very dodgy AT NEW statement, which didn’t work particularly well in a lot of circumstances. In fact, the AT NEW statement was so unpredictable that it was wiser to avoid it altogether and use other means to process related chunks from your internal table.

To be specific, if you said AT NEW MONSTER in a loop, you would expect the code within that block to execute when the monster changed. However, it sometimes wouldn’t execute, depending on how the table was sorted or if Jupiter was in alignment with Neptune or if the chicken had jumped up and down three times at midnight and laid three addled eggs.

To avoid your program having to depend on such astrological or chicken-related factors, ABAP 7.4 announced good news: a GROUP BY option has been added to looping at internal tables.

To understand what this means, let’s look at an example. Say that you have a huge table of monsters. There are monsters of various types: some mad, some not. You only want to process one combination of monster type/madness at a time. Somehow, you need to aggregate such records, do something with the result, and move on to the next combination. To do so, you need some sort of nested loop processing.

The usual way to do this is to build a smaller table of all possible combinations found within the main table, do an outer loop on that small table, and for each row of that outer table do an inner loop on the main table, looking for records relating to the combination in the outer loop. There are ways to do this well—for example, by making sure the inner table is a SORTED table—but it’s still a lot of effort.

Fortunately, GROUP BY saves the day. Listing 3.49 demonstrates how applying the GROUP BY construct when looping over an internal table lets you look at the aggregate records for each combination one at a time, without having to go through the agony of nested loops. On each pass through the table, you’re handed the aggregated data on a plate.

```
DATA(table_of_monsters) = VALUE tt_monsters(
( monster_number = '1' model = 'BTNK' no_of_heads = 3 )
( monster_number = '2' model = 'BTNK' no_of_heads = 4 )
( monster_number = '3' model = 'BTNK' no_of_heads = 2 )
( monster_number = '4' model = 'ISDD' no_of_heads = 1 )
( monster_number = '5' model = 'ISDD' no_of_heads = 1 )
).

LOOP AT table_of_monsters ASSIGNING FIELD-SYMBOL(<monster_details>)
  GROUP BY ( model = <monster_details>-model
    is_it_crackers = zcl_4_monster_model=>is_mad( <monster_details>-monster_number )
  )
  ASSIGNING FIELD-SYMBOL(<monster_group_record>);
  CHECK <monster_group_record>-is_it_crackers = abap_true.
  CLEAR monster_sub_set.
  LOOP AT GROUP <monster_group_record> ASSIGNING FIELD-SYMBOL(<bonkers_monsters>).
    monster_sub_set = VALUE #( BASE monster_sub_set ( <bonkers_monsters> ) ).
  ENDDOLOOP;
  CLEAR total_heads.
  LOOP AT monster_sub_set ASSIGNING FIELD-SYMBOL(<sub_set_record>).
    total_heads = total_heads + <sub_set_record>-no_of_heads.
  ENDDOLOOP;
  WRITE:/ 'Bonkers Monsters of Type',<monster_group_record>-model,' have ',
    total_heads,' heads'.
ENDLOOP.
```

* In this example we are trying to prove that bonkers monsters with bolts through their necks have more heads.
* per monster on average compared to bonkers monsters who like to ice skate.
* This is most likely a 100% correlation with the business problem you are trying to solve at work right at this moment.

```
TYPES: tt_monsters TYPE STANDARD TABLE OF z4t_monster_head
  WITH DEFAULT KEY.

DATA: monster_sub_set TYPE tt_monsters,
    total_heads TYPE i.
```

First, the database read is simulated to fill a table of monsters with three “bolts through neck” monsters and two ice skaters, each of which has one or more heads. The business requirement is going to be an aggregated list of the total number of heads, summarized by type of monster and excluding all monsters who are not mad.
Next is the LOOP AT/GROUP BY construct. What happens here is that every line of the table is looped through before moving onto to the line starting with CHECK. In debugging, you can see that the IS_IT_MAD method is called five times, once for each row in the table. To simplify matters, all the monsters are mad in this example.

Now, there’s some sort of invisible nebulosity table with two rows, two columns long, which has been built up by a procedure analogous to COLLECT. The first row says ISDD/X and the second row says ISDD/x. This is the table of possible combinations, the table that will be looped through into the <MONSTER_GROUP_RECORD> work area.

A check is performed to ensure we’re dealing with a combination that involves mad monsters. Once that’s determined, the time has come to create a table that’s a subset of the main table, a subset that only includes monsters that match the current combination.

The processing block starting with LOOP AT GROUP performs this task. The individual records from the main table that match the current combination are looped through into the <BONKERS_MONSTERS> work area. At this point, you could just add <BONKERS_MONSTERS>-PEOPLE_SCARED to the TOTAL_SCARED variable, but to prove a point we’re going to build up a subset table.

First, we’ll look at the line that reads as follows:

```
monster_sub_set = VALUE #( BASE monster_sub_set ( <bonkers_monsters> ) ) .
```

This could be translated as “append to table MONSTER_SUB_SET the <BONKERS_MONSTERS> work area.”

Now we have a table containing a subset of the main table, and we loop through those records, adding up the total number of heads of the current type of monster and outputting the result.

The result is two written lines saying that (1) the mad monsters with bolts through their necks have nine heads in total and (2) the mad ice skating dead have two heads in total.

**Note**

As mentioned, we didn’t really need to create a subset table in this example, but it’s there to show that it’s possible.

At this point, you might be thinking this looks just like a geometric loop—but the inner loop is only processed once for each group of similar items that the outer loop finds. Therefore, the runtime only increases in a linear fashion, which means that your program isn’t going to get into the situation in which it will be fine for a small number of records but will time out when there’s a lot of data.

### 3.5 Internal Tables

#### 3.5.9 Extracting One Table from Another

There are two new ways to extract one internal table from another that were introduced with 7.4, and both use the constructor operator FILTER. The next subsection will talk about using the FILTER operator first with conditional logic and then as a FOR ALL ENTRIES operation on an internal table.

**Filter with Conditional Logic**

To understand the process of extracting one table from another in ABAP 7.4, return to the monsters example. Once again, you have a big table of all the monsters, and you want to extract a smaller internal table with just the averagely mad monsters. Normally, you would just loop through the big table and append lines to your new table, as shown in Listing 3.50.

```
DATA: "Source Table
    all_monsters
    TYPE SORTED TABLE OF z4t_monster_head
    WITH NON-UNIQUE KEY monster_number
    WITH NON-UNIQUE SORTED KEY bonkers_ness
    COMPONENTS sanity_percentage,
    "Target Table
    averagely_mad_monsters
    TYPE STANDARD TABLE OF z4t_monster_head,
    _averagely_mad_monster LIKE LINE OF averagely_mad_monsters.

    "Extract Source to Target
    LOOP AT all_monsters ASSIGNING FIELD-SYMBOL(<monster_record>)
    WHERE sanity_percentage < 75.
    CLEAR _averagely_mad_monster.
    MOVE-CORRESPONDING <monster_record> TO _averagely_mad_monster.
```

In the SQL version of GROUP BY, you can only use column names. Because the internal table version of GROUP BY is processed wholly within ABAP, you can do all sorts of groovy things in addition—comparisons, method calls, and the like—which is why the example included a method call to IS_IT_MAD.

This takes quite a bit of experimentation before you get comfortable with it. The most difficult part, as with all these new constructs, is marrying new abilities to real-world problems. (Of course, some people might say that summarizing open items per customer from an internal table filled with BSID values is more realistic than summarizing head numbers of different types of monsters, but what do they know?)
As of ABAP 7.4 (SP 8), you can do the same thing by using the constructor operator `FILTER` in the fashion shown in Listing 3.51.

```abap
DATA( averagely_mad_monsters ) =
  FILTER( all_monsters USING KEY bonkers_ness
    WHERE sanity_percentage < CONV( 75 ) ).
```

Listing 3.51 Extracting One Table from Another in 7.40

Therefore, in 7.4 you can do something you couldn’t do before—but there’s a caveat. The problem is that the `FILTER` operation introduced in 7.4 only works if the large table has either a hashed or sorted key. In the preceding example, `ALL_MONSTERS` has a sorted key on `sanity_percentage`, so the caveat doesn’t apply. (If the internal table `ALL_MONSTERS` didn’t have a hashed or sorted index, then it wouldn’t be such a huge problem; remember that ever since 7.02, internal tables can have several secondary keys, which can be hashed or sorted.)

### FILTER Used as FOR ALL ENTRIES on an Internal Table

When reading from the database into an internal table, if you don’t want all of your data in one massive table or you can’t merge all the tables together by inner joins, the solution has always been—since the year 2000 and still to this day—to perform `FOR ALL ENTRIES`, as shown in Listing 3.52.

```abap
SELECT *
  FROM z4t_deliveries
  INTO CORRESPONDING FIELDS OF TABLE monster_deliveries
     FOR ALL ENTRIES IN all_monsters
     WHERE monster_number = all_monsters-monster_number.
```

Listing 3.52 FOR ALL ENTRIES during Database Read

If earlier in the program you already had read the entire `monster_deliveries` table from the database into an internal table for some reason and wanted to create a subset just for the monsters currently in `ALL_MONSTERS`, then you could now run the equivalent of `FOR ALL ENTRIES`, but on an internal table rather than the database (see Listing 3.53).

```abap
DATA( deliveries_our_monsters ) =
  FILTER( monster_deliveries IN all_monsters
    WHERE monster_number = monster_number ).
```

Listing 3.53 FOR ALL ENTRIES on Internal Table

Once again, the filter table (`ALL_MONSTERS`, in this case) must have a sorted or hashed key that matches what we’re searching for (the monster number, in this case).

### 3.5.10 Virtual Sorting of Internal Tables

What in the world is virtual sorting? This feature (which arrived in ABAP 7.52) is going to be quite difficult to get your head around. In fact, it could be described as a solution looking for a problem. Nonetheless, even if I can’t think of a practical use for it, it’s more than possible that you’re currently sitting at work grappling with a seemingly insoluble problem that it might just solve. It’s almost impossible to explain without an example, so let’s walk through one.

Say you have an existing report that’s displayed in the form of an ALV tree. The header table contains a list of customer disputes, cases in which a customer has complained to the baron that the monster didn’t scare the peasants enough and they want a partial refund. In the early days, the baron used to respond by sending round a monster to kill and eat the complaining customer, but this didn’t do much in the way of encouraging repeat business, so these days he often raises credit notes for valid claims.

Thus, in the ALV dispute report, each dispute line can be expanded to show the credit note raised against the dispute or an entry saying the dispute has been rejected. So there are two internal tables, a header one for the disputes and an item one linking disputes to credit notes. The two tables are the same size, but the data is split so that all columns can be viewed on one screen without scrolling to the right. Everything had worked fine for ages; everything in the garden was rosy.

Then last week the chief accountant came to Inga, the head developer. In this case, the chief accountant told her that he was investigating some sort of fraud possibly being perpetrated against the baron and needed an ABAP2XLXS export of the data in the dispute report. All he needed was a list of the dispute details, sorted so that the ones relating to high-value credit notes were at the top.

This initially appeared to be a simple task (adding a spreadsheet export option to a report is so easy), but the fact that there are two tables complicates matters. Should Inga make it so that each table is on a separate worksheet? That would work, but then the user would have to do some manual work to link the two together, thus rather defeating the purpose of computer automation.

After mulling it over, Inga realized she wanted to keep the header table and the item table exactly as they are—sorted by `DISPUTE` number—so that the report would look like it always did. To do so, she would have to create a new table with the data from the dispute plus the credit note value, sort that, and use it for export to a spreadsheet.

That isn’t actually too difficult, but just for giggles, let’s do it using virtual sorting. In Listing 3.54, you’ll see that there’s a new method of `CL_ABAP_ITAB_UTILITIES`, into which are passed references to the two source tables (they have to be the same size, and you
can pass in as many source tables as you want). For each table, you pass in the sort criteria. Ascending is the default, as usual; there’s a parameter to be filled if you want the sort to be descending.

```
DATA(dispute_indexes) = cl_abap_itab_utilities->virtual_sort(
  in_virtual_source =
  VALUE #(
    ( source = REF #( credit_note_table )
    components =
    VALUE #(
      ( name = 'credit_value'
      descending = abap_true )
    )
    ( source = REF #( dispute_table )
    components =
    VALUE #(
      ( name = 'dispute_number' )
    )
  )
).
```

Listing 3.54 Virtual Sorting

The result, DISPUTE_INDEXES, is a table that’s one-column wide and contains what the index numbers would have been had you actually created a new table with the columns from both tables and then sorted it by value ascending and dispute number ascending.

Then a new table is declared for the purposes of output to Excel, and it gets filled using the indexes, as demonstrated in Listing 3.55.

```
DATA top_disputes TYPE STANDARD TABLE OF l_typ_disputes.

LOOP AT dispute_indexes ASSIGNING FIELD-SYMBOL(<index>),
  APPEND dispute_table[<index>] TO top_disputes.
ENDLOOP.
```

Listing 3.55 Using Virtual Sort Result

It’s possible to do the previous two steps in one big statement that’s so complicated that no one could ever be expected to understand it—but we’re not going to demonstrate that.

To my mind, this isn’t an easier way to achieve a task, just a different way. Nonetheless, now you know about it, just in case it’s exactly the sort of thing you’re looking for.

### 3.6 Object-Oriented Programming

SAP introduced the concept of OO programming in ABAP in the year 2000. Since then, it’s been the recommended method of programming. This section will describe the new features of ABAP that relate to OO programming.

Before we jump in, I have a burning need to mention that in ABAP 7.5, Transaction SE24 (Maintain Class) has been changed so that the full width of the screen is used. This is such a simple fix, but so useful, and long, long overdue. Maybe one day the same streak of common sense will be applied to SE93, which is one of the most cramped screens around—and, as of ABAP 7.55, still hasn’t been fixed!

#### 3.6.1 Upcasting/Downcasting with CAST

In OO programming, a downcast is a process in which you turn a generic object, like a monster, into a more specific object, like a green monster. An upcast is the reverse. This functionality has been available in ABAP for a long time, but it gets a lot easier in 7.4.

To take a non-monster-related example for once, consider a situation in which you need to get all the components of a specific dictionary structure. Listing 3.56 shows how you would do this prior to ABAP 7.4. First, call a method of CL_ABAP_TYPEDESCRIPTOR to get metadata about a certain structure. But to get the list of components of that structure into an internal table, you need an instance of CL_ABAP_STRUCTDESCR; this is a subclass of CL_ABAP_TYPEDESCRIPTOR. Thus you need to perform a downcast to convert the instance of the parent class into an instance of the subclass.

```
DATA structure_description TYPE REF TO cl_abap_structdescr.
  structure_description += cl_abap_typedescr->describe_by_name( 'Z2SP_MONSTER_HEADER_D' ).
  DATA structure_components TYPE abap_compsdescr_tab.
  structure_components = structure_description->components.
```

Listing 3.56 Components of Specific Dictionary Structure without CAST

In 7.4, you can do this all in one line by using the CAST constructor operator (see Listing 3.57).

```
DATA(structure_components) = CAST cl_abap_structdescr(cl_abap_typedescr->describe_by_name( 'Z2SP_MONSTER_HEADER_D' ))->components.
```

Listing 3.57 Components of Specific Dictionary Structure with CAST

The code in Listing 3.56 and Listing 3.57 performs exactly the same function, but in the latter case you no longer need the STRUCTURE_DESCRIPTION helper variable and you also don’t need the line in which you define the STRUCTURE_COMPONENTS type.

#### 3.6.2 Finding the Subclass of an Object Instance

In many other programming languages, it’s possible to work out, given an object reference, what precise subclass that instance is. Prior to 7.5, the ABAP team at SAP resisted this, but after unceasing demand from the online SAP Community, SAP provided the new IS_INSTANCE_OF statement.
Just to rain on SAP’s parade even more—and this is very cruel, considering that it only added this new feature due to popular demand—some purists would say that any subclass should be able to impersonate its parent without any program knowing the difference, and thus a calling program wouldn’t need to know the exact subclass. However, the ABAP world isn’t as pure as everyone would like, and sometimes knowing this information is actually quite useful—as in the following example.

In Chapter 10 on ALV, there’s an example in which we try to get an ALV grid reference and try one subclass after another until the assignment succeeds (which we repeat here in Listing 3.58).

```abap
DATA: full_screen_adapter TYPE REF TO cl_salv_fullscreen_adapter,
  container_adapter TYPE REF TO cl_salv_grid_adapter.
TRY.
  "Presume full screen mode (No Container)
  "fullscreen Adapter (Down Casting)
  "Target FULL SCREEN_ADAPTER = CL_SALV_FULLSCREEN_ADAPTER
  "CL_SALV_FULLSCREEN is a subclass of CL_SALV_ADAPTER
  full_screen_adapter ?= io_salv_adapter.
  "Get the Grid
  ro_alv_grid = full_screen_adapter->get_grid( ).
CATCH cx_sy_move_cast_error.
  "Must be in container mode
  "CL_SALV_GRID_ADAPTER is a subclass of CL_SALV_ADAPTER
  container_adapter ?= io_salv_adapter.
  ro_alv_grid = container_adapter->get_grid( ).
ENDTRY.
```

**Listing 3.58** Trying to Find Subclass before 7.5

In 7.5, life becomes much easier, as shown in Listing 3.59. By using the IS_INSTANCE_OF construct, the purpose of the code becomes much clearer to the reader.

```abap
IF io_salv_adapter IS INSTANCE OF cl_salv_fullscreen_adapter.
  full_screen_adapter ?= io_salv_adapter.
  ro_alv_grid = full_screen_adapter->get_grid( ).
ELSEIF io_salv_adapter IS INSTANCE OF cl_salv_grid_adapter.
  container_adapter ?= io_salv_adapter.
  ro_alv_grid = container_adapter->get_grid( ).
ENDIF.
```

**Listing 3.59** Trying to Find Subclass in 7.5

The same task can be achieved in a slightly different way by using the TYPE OF construct in conjunction with CASE. In the code in Listing 3.60, the exact subclass is determined, and the respective branch of the CASE statement ensures that the created instance has the correct subclass by using another new construct: INTO DATA.

```abap
CASE TYPE OF io_salv_adapter.
  WHEN TYPE cl_salv_fullscreen_adapter
    INTO DATA(full_screen_adapter2).
  WHEN TYPE cl_salv_grid_adapter
    INTO DATA(container_adapter2),
    ro_alv_grid = container_adapter2->get_grid( ).
  WHEN OTHERS.
    RETURN.
ENDCASE.
```

**Listing 3.60** Another Way to Find Subclass in 7.50

The way you have to write the code in examples such as the one in Listing 3.60 isn’t particularly clear; the phrase INTO DATA doesn’t read much like an English sentence and thus could be confusing. Nonetheless, you need to know that the option is available.

### 3.6.3 CHANGING and EXPORTING Parameters

In ABAP, a functional method has until now been defined as a method with one returning parameter and zero to many importing parameters, such as the following:

```abap
monster_header = monster->get_details( monster_number ).
```

Many ABAP programmers liked the fact that they could put the result variable at the start rather than having to put that variable in an EXPORTING parameter. But they wanted to be able to have CHANGING and EXPORTING parameters as well—that is, to have their cake and eat it too.

In 7.40, SAP has waved its magic wand, and now you can have it both ways. An example is shown in Listing 3.61.

```abap
* Local Variables
DATA: monster_number TYPE zde_monster_number VALUE '0000000001',
  something_spurious TYPE string,
  something_unrelated TYPE string.

DATA(monster_header_record) = lcl_monster=>get_details( ).

EXPORTING
  id_monster_number = monster_number
  sd_monster_spurious = something_spurious
  sd_monster_unrelated = something_unrelated.

CHANGING
  c'd_monster_unrelated = something_unrelated
```

**Listing 3.61** CHANGING and EXPORTING Parameters

There is no doubt that many people will be happy with this, but purists who are used to other languages will be horrified. (Although I don’t feel quite that strongly, I can see their point.) Good OO design leads you toward small methods that do one thing, and
the one thing for functional methods is to output one result. If a functional method
suddenly starts giving you back all sorts of other exporting parameters and changes
something else, then the method is clearly doing more than one thing, and that’s prob-
ably bad design. (For example, there are methods designed to return four values of a
polynomial equation, and you could use the new design to put the first value in the
RETURNING parameter and the last three values in EXPORTING parameters, but that seems
a bit silly. You could just return a structure of four values instead.) Nonetheless,
because this is a new feature of 7.4 that you might occasionally find useful, it was
important to mention it here.

Fun Fact about Functional Methods
If you have a functional method that returns a structure, but you only want one field of
that structure, you can write the following code:

```abap
DATA(sanity) = lcl_monster=>get_details( monster_number)-sanity_percentage.
```

The result will be a `sanity` variable typed as `Z4DE_MONSTER_SANITY_PERCENTAGE`. This is
because both at compile time and at runtime, a call to a functional method is interpreted as a variable of whatever type it returns.

3.6.4 Changes to Interfaces

This section discusses how SAP has tried to take some of the pain out of your daily usage
of interfaces in OO programming in 7.4. As you know, an interface is a collection of
declaration and method names and signatures. If a class implements any given
interface, then it has to redefine all the interface methods. This is all good, but prior to
7.4 the problem was that some standard interfaces had a really big list of methods, only
some of which were relevant, so you had to go through the irrelevant methods and
redefine them to have blank implementations.

As of 7.4, if you create an interface and think that some of the methods might not be
needed by all classes that implement the interface, then you can say so in the interface
definition, as shown in Listing 3.62.

```abap
INTERFACE scary_behavior.
METHODS: scare_small_children,
  sells_mortgages DEFAULT FAIL,
  hide_under_bed DEFAULT IGNORE,
  is_fire_breather
  DEFAULT IGNORE
  RETURNING rf_yes_it_is TYPE abap_bool.
ENDINTERFACE. "Scary Behavior
```

Listing 3.62 Defining Interface with Optional Methods

This is an interface all monster classes should implement. Naturally, all monsters should
be able to scare children—so don’t put any additions after that method definition. This
means that each class implementing that interface is forced to redefine the method by
the syntax check. On the other hand, most monsters will not sell mortgages (just the
worst of the worst monsters), so don’t force all the classes to implement this method.
Because you’ve added DEFAULT FAIL, if a program using an instance of a monster that
implements this interface tries to make the monster sell mortgages, and the method
hasn’t been implemented, then a runtime error occurs (CX_SY_DYN_CALL_ILLEGAL_METHOD).
Similarly, don’t force all monster classes to hide under beds; obviously, the ones that
are a thousand feet tall have problems in this area. By adding DEFAULT IGNORE to the end
of the definition, we can make sure these classes aren’t forcibly implemented. If the
program tells such a monster to hide under the bed, then nothing will happen—just as
if a call had been made to an implemented method with no lines of code inside it.

In the same way, not all monsters breathe fire. For the ones that do, the IS_FIRE_BREATHER method can be implemented to return ABAP_TRUE. If the method is not imple-
mented in any given monster class, then the DEFAULT IGNORE addition is used, and the
RETURN parameter will bring back an initial value, which in this case is ABAP_FALSE.

3.7 Search Helps

Search helps are one of the strengths of the SAP system. You can define one and attach
it to a data element, and then all throughout the system wherever that data element is referenced, an [F4] dropdown of possible values is available instantly. In release 7.4, things even have become slightly better, and this section will explain how.

3.7.1 Predictive Search Helps

If you’ve read anything in the IT media in the last few years, then you’ll be sick to death
of writers saying that business users now expect at work the same level of usability
they get in their spare time. At this point, you’ve probably seen children with iPads;
they swipe away at the screen, calling up this and that, already more adept at this than
many adults. If that’s the level of user-friendliness children are exposed to, then what
are they going to expect from enterprise software like SAP when they grow up and get
jobs? What would someone who had just left school expect from an SAP system they
were being shown how to use on their first day of work?

For one thing, they would expect that when they start typing something in a search
field—a customer or material name, for example—after a letter or two has been typed,
the dropdown of possible values will appear for them to choose from. That’s
what happens on Google and many other websites, and you can see why recent stu-
dents would be shocked when this doesn’t happen on an SAP screen.
The good news is that if your SAP system is 7.4 SP 3 or above and SAP GUI is 7.31 patch level 5 or above, then you’ll find that such predictive search helps are now possible in SAP. Half of the functionality is in the backend, and half is in SAP GUI, which is why you need both to be on the correct level. (In fact, you really need to be on 7.4 SP 6 for this to work automatically; otherwise, you have to fluff around manually with the CL_DSH_DYNPRO_PROPERTIES class, as described in SAP Note 1861491.)

Assuming you’re on the right version, open Transaction SE11 and go into the Search Help option about halfway down the screen. There, you’ll see the Data Collection area (see Figure 3.2).

![Figure 3.2 Search Help Definition](image_url)

In release 7.4, directly above the Search Help Exit box is a new box called Enhanced Options, in which you have two checkboxes and an input field. The second checkbox and the input field are to do with fuzzy text searches on an SAP HANA database and are grayed out, but the first checkbox says Autosuggest in Input Fields.

If you select that checkbox, then your search help will magically start behaving differently—not quite like Google, but a popup list of candidates will appear as the user types, which is a step up from before.

### 3.7.2 Search Help in SE80

You may have noticed, and been bothered by, the fact that when you’re in SE80 and can’t remember the name of your program, you can’t just press \( F4 \) and see the result list. You have to press the dropdown arrow to the right of the input field, which presumably is why that dropdown arrow’s there in the first place. (After all, if \( F4 \) worked, then you wouldn’t need a separate dropdown arrow.)

So you can imagine my delight when I was playing around in one of the latest SAP releases and was in SE80 and had typed in half of my program name and pressed \( F4 \) by reflex. I hadn’t even put in a wildcard asterisk, but what should I see but a dropdown list of results (see Figure 3.3).

Mere words cannot explain how happy this made me. It must be true that little things please little minds. It turns out that this function was available as of release 7.31—but not everyone knows about it, so it’s worth calling out here.

![Figure 3.3 F4 Search Help in SE80: Working at Long Last](image_url)

### 3.8 Assorted

Much as I try to fit all the new ABAP topics into round holes, there are some square pegs that don’t fit neatly anywhere but are still worth a mention. We’ll look at those square pegs in this section.

#### 3.8.1 Unicode

One change in 7.5 that doesn’t fit into any category is that now if you create a program or class or what have you and check the Unicode checkbox, the system lets you create the object, but then you get a fatal error in the syntax check until you switch it back off again. I can see why you need the box, to change the setting for old programs, but if I ruled the world, I’d have the system not even let you create the object unless the box was checked—and likewise for Fixed Point Arithmetic, which the 7.5 system doesn’t complain about at all if it’s off.

#### 3.8.2 ABAP Language Versions

When you create a program, one setting you probably never even look at is the ABAP Language Version in the attributes section of an executable program or a class. It defaults to \( X \) (Standard ABAP Unicode) and you think nothing of it. In fact, if you were to select the dropdown, you would see a whole bunch of options, as shown in Figure 3.4.
3.8 Assorted

Apart from the two mentioned just now, the other deprecated statements are as follows:

- **AT END, AT FIRST, AT LAST, AT NEW, ENDT**
  These statements have never worked. I have not used them for 20 years.

- **ADD, DIVIDE, MULTIPLY, SUBTRACT**
  I will actually miss these. I thought they made the code look clearer. I think \texttt{ADD 1 TO MONSTER\_COUNT reads more like plain English than the new MONSTER\_COUNT ++1 equivalent, though the latter does sound like something from the novel 1984 mentioned at the start of the chapter.}

- **RAISE, MESSAGE RAISING**
  We’re talking about “classical” exceptions here. Only class-based exceptions will be allowed going forward.

- **FORM, ENDFORM, PERFORM**
  This news will give old-school ABAPers who have been doing things the exact same way for 20 years a heart attack. No more \texttt{FORM} routines! That’s almost as bad as having to stop riding horses and drive cars instead!

### 3.8.4 Clean ABAP

Let’s discuss three open-source projects, all to do with improving the quality of your ABAP code. All three live on GitHub, and you’ll find links to all three in the Recommend Reading section at the end of the chapter.

- **Clean ABAP**
  First up is the most wonderful ABAP-language open-source project ever created: Clean ABAP, which was started by SAP but which anyone can contribute to. This is a set of ABAP programming guidelines based on the Clean Code book by Robert Martin. That was one of the best programming books ever written, but a lot of ABAP people ignored it because all the examples were in Java. This is not to do with functional correctness like traditional ABAP Test Cockpit checks but rather clarity: the easier your code is to understand, the easier (and cheaper) it is to make corrections and changes.

  This takes the form of an ABAP programming style guide. In essence, there are about 10 billion rules, with explanations of why each rule is a Good Thing, coupled with good and bad examples. Naturally, not everyone will agree with everything (monsters don’t generally like being clean, for example), but most programmers will agree with the vast majority of the suggested guidelines.

- **ABAP Test Cockpit checks**
  In addition, you can download from the GitHub URL special ABAP Test Cockpit checks created by SAP that you can add into your default Code Inspector check variant, so you can see how “clean” your code really is.
3 New Language Features in ABAP

If that wasn’t enough, there’s yet another open-source project in this area—this time to do with ABAP code reviews. You probably already have a formal or informal process whereby one of your colleagues looks over your code before it goes to QA; this project seeks to compile a list of best practices (and helpful tools) to boost your efforts in this area.

3.9 Summary

In this chapter, you read about the somewhat radical changes that have been introduced in the ABAP language in recent years, starting with 7.02 but most notably in releases 7.4 and above. This chapter cataloged the vast array of new language constructs that have been introduced into ABAP and showed examples of where they might be useful in your day-to-day work, broken into several categories.

Now that you’ve learned about the development environment (Eclipse), the version-management control coming in the future (abapGit), and the increased capabilities of the language you program in, it’s time to move on to how to handle cases where everything falls in a heap—in other words, exceptions.

Recommended Reading

- Clean ABAP style guide
  https://github.com/SAP/styleguides/blob/main/clean-abap/CleanABAP.md
  (Klaus Haeuptle et al.)

- Clean ABAP ATC checks
  https://github.com/SAP/code-pal-for-abap

- ABAP code review guide
  https://github.com/SAP/styleguides

- Clean ABAP
  (Klaus Haeuptle et al., SAP PRESS, 2021)
Contents

Acknowledgments ................................................................................................................ 19
Introduction ...................................................................................................................... 21

1 Integrated Development Environment .............................................................................. 31

1.1 Installation .......................................................................................................................... 33
  1.1.1 Installing Eclipse ..................................................................................................... 33
  1.1.2 Installing SAP-Specific Add-Ons ........................................................................ 36
  1.1.3 Connecting Eclipse to a Backend SAP System ............................................... 37
  1.1.4 Upgrading Eclipse ................................................................................................... 39

1.2 ABAP-Specific Features ....................................................................................................... 39
  1.2.1 Initial Tour and Basic Tasks ................................................................................. 40
  1.2.2 Working on Multiple Objects at the Same Time .......................................... 44
  1.2.3 Creating a Method from the Calling Code ..................................................... 45
  1.2.4 Extracting a Method .............................................................................................. 49
  1.2.5 Refactoring: Moving Methods and Attributes ............................................. 54
  1.2.6 Deleting Unused Variables .................................................................................. 54
  1.2.7 Creating Instance Attributes and Method Parameters ............................. 56
  1.2.8 Quick Fixes for Classes .......................................................................................... 56
  1.2.9 Extracting Conditional Logic ............................................................................... 57
  1.2.10 Seeing Message Details ........................................................................................ 59
  1.2.11 Renaming Repository Objects ............................................................................ 59
  1.2.12 Magic Numbers ....................................................................................................... 61
  1.2.13 ABAP 7.5+ Features ................................................................................................ 61

1.3 Eclipse-Specific Features .................................................................................................... 65
  1.3.1 Neon (2016) .............................................................................................................. 66
  1.3.2 Oxygen (2017) ......................................................................................................... 67
  1.3.3 Photon (2018) .......................................................................................................... 68
  1.3.4 Eclipse 2020-03 ....................................................................................................... 68

1.4 Testing and Troubleshooting .............................................................................................. 70
  1.4.1 Unit Testing .............................................................................................................. 70
  1.4.2 Debugging ................................................................................................................ 71
  1.4.3 Dynamic Log Points ............................................................................................... 74
  1.4.4 Runtime Analysis .................................................................................................... 76

1.5 Customization Options with User-Defined Plug-Ins ....................................................... 77
  1.5.1 Favorites List ............................................................................................................. 78
Contents

1.5.2 Continuous Integration ......................................................... 80
1.5.3 Code Insight ........................................................................ 83
1.5.4 Custom Quick Fixes ............................................................. 83
1.5.5 Custom Quick Fixes for ABAP Test Cockpit ......................... 85
1.6 The Future of IDEs for ABAP Development ......................... 85
1.6.1 SAP Web IDE ................................................................. 85
1.6.2 SAP Business Application Studio ........................................ 86
1.6.3 Visual Studio Code ............................................................ 89
1.7 Summary ................................................................................. 96

2 abapGit ................................................................. 99

2.1 Theory ...................................................................................... 100
2.2 Installation ............................................................................... 101
2.2.1 Installing the abapGit Repository in Your SAP System .......... 101
2.2.2 Keeping Your abapGit Version Up to Date ......................... 107
2.2.3 Watching the abapGit Repository ........................................ 109
2.3 Storing and Moving Objects .................................................. 110
2.3.1 abapGit versus SAPlink ..................................................... 111
2.3.2 Using Online Repositories ................................................ 111
2.3.3 Using Offline Repositories ................................................ 117
2.4 Dependency Management ...................................................... 122
2.4.1 APACK: Theory ............................................................... 123
2.4.2 APACK: Installation .......................................................... 124
2.4.3 APACK: Example ............................................................... 124
2.5 Branching ................................................................................ 125
2.5.1 Project Collaboration: Sharing Solutions ......................... 126
2.5.2 Production Support ............................................................ 133
2.5.3 Utopian Dream ................................................................. 141
2.6 abapGit for Customizing ......................................................... 141
2.7 Summary ................................................................................. 142

3 New Language Features in ABAP ............................................... 145

3.1 Declaring and Creating Variables ........................................... 147
3.1.1 Omitting Data Type Declarations ...................................... 147

3.2 String Processing ................................................................. 155

3.3 Calling Functions .................................................................... 156
3.3.1 Avoiding Type Mismatch Dumps when Calling Functions .... 156
3.3.2 Using Constructor Operators to Convert Strings ............... 158
3.3.3 Functions Expecting TYPE REF TO DATA .......................... 159

3.4 Conditional Logic ................................................................... 160
3.4.1 Omitting ABAP_TRUE ...................................................... 161
3.4.2 Using XSDBOOL as a Workaround for BOOLC ................. 162
3.4.3 The SWITCH Statement as a Replacement for CASE ........ 164
3.4.4 The COND Statement as a Replacement for IF/ELSE ......... 165

3.5 Internal Tables ........................................................................ 167
3.5.1 Table Work Areas ............................................................ 167
3.5.2 Reading from a Table ........................................................ 169
3.5.3 CORRESPONDING for Normal Internal Tables ................. 170
3.5.4 MOVE-CORRESPONDING for Internal Tables with Deep Structures .............................................................................. 172
3.5.5 Dynamic MOVE-CORRESPONDING ................................ 175
3.5.6 New Functions for Common Internal Table Tasks .............. 177
3.5.7 Internal Table Queries with REDUCE ............................... 179
3.5.8 Grouping Internal Tables .................................................. 181
3.5.9 Extracting One Table from Another ................................... 185
3.5.10 Virtual Sorting of Internal Tables ...................................... 187

3.6 Object-Oriented Programming .............................................. 188
3.6.1 Upcasting/Downcasting with CAST .................................. 189
3.6.2 Finding the Subclass of an Object Instance ......................... 189
3.6.3 CHANGING and EXPORTING Parameters ....................... 191
3.6.4 Changes to Interfaces ....................................................... 192

3.7 Search Helps .......................................................................... 193
3.7.1 Predictive Search Helps ................................................... 193
3.7.2 Search Help in SE80 .......................................................... 194

3.8 Assorted .................................................................................. 195
3.8.1 Unicode ............................................................................ 195
3.8.2 ABAP Language Versions ............................................... 195
8 Business Logic Using the ABAP RESTful Application Programming Model

8.1 ABAP RESTful Application Programming Model versus BOPF

8.1.1 Business Definition Language

8.1.2 Business Objects as First-Class Citizens

8.2 Coding Business Object CDS Entities

8.2.1 Coding Data Definitions

8.2.2 Coding CDS Projections

8.2.3 Coding Metadata Extensions

8.3 Coding Behavior Definitions and Projections

8.4 Coding Behavior Implementations

8.4.1 Creating a Behavior Class

8.4.2 Creating or Changing Objects

8.4.3 Locking Objects

8.4.4 Performing Authority Checks

8.4.5 Determinations: Deriving Values from Other Values

8.4.6 Validations: Checking Data Integrity

8.4.7 Actions: Responding to User Input

8.4.8 Feature Control

8.4.9 Saving the Changed Business Objects

8.4.10 Tracking Changes in ABAP RESTful Application Programming Model Objects

8.5 Calling CRUD Operations from ABAP

8.6 Summary

9 Service Layer

9.1 What Is SAP Gateway?

9.2 Transaction SEGW Service Layer: Manual Creation

9.2.1 Configuration

9.2.2 Coding

9.3 Transaction SEGW Service Layer: Automatic Creation

9.3.1 Creating an SAP Gateway Service by Pulling from a CDS View

9.3.2 Creating an SAP Gateway Service by Pushing from a CDS View

9.4 ABAP RESTful Application Programming Model Service Layer: Manual Creation

9.4.1 Creating a Service Definition

9.4.2 Creating a Service Binding

9.4.3 Creating Automated Unit Tests for the OData Service

9.5 ABAP RESTful Application Programming Model Service Layer: Automatic Creation

9.5.1 Making Sure the RAP Generator Is Available in Your System
## 10 ALV SALV Reporting Framework

### 10.1 Getting Started
- 10.1.1 Defining an SALV-Specific (Concrete) Class
- 10.1.2 Coding a Program to Call a Report

### 10.2 Designing a Report Interface
- 10.2.1 Report Flow Step 1: Creating a Container (Generic/Optional)
- 10.2.2 Report Flow Step 2: Initializing a Report (Generic)
- 10.2.3 Report Flow Step 3: Making Application-Specific Changes (Specific)
- 10.2.4 Report Flow Step 4: Displaying the Report (Generic)

### 10.3 Adding Custom Command Icons with Programming
- 10.3.1 Creating a Method to Automatically Create a Container
- 10.3.2 Changing ZCL_BC_VIEW_SALV_TABLE to Fill the Container
- 10.3.3 Changing the INITIALIZE Method
- 10.3.4 Adding the Custom Commands to the Toolbar
- 10.3.5 Sending User Commands from the Calling Program
- 10.3.6 Adding Separators

### 10.4 Editing Data
- 10.4.1 Creating a Custom Class to Hold the Standard SALV Model Class
- 10.4.2 Changing the Initialization Method of ZCL_BC_VIEW_SALV_TABLE
- 10.4.3 Adding a Method to Retrieve the Underlying Grid Object
- 10.4.4 Changing the Calling Program
- 10.4.5 Coding User Command Handling

### 10.5 Handling Large Amounts of Data with CL_SALV_GUI_TABLE_IDA
- 10.5.1 Basic Example
- 10.5.2 Complex Example

### 10.6 Open-Source Fast ALV Grid Object

### 10.7 Making SAP GUI Look Like SAP Fiori

### 10.8 Summary

## 11 ABAP2XLSX and Beyond

### 11.1 The Basics
- 11.1.1 How XLSX Files Are Stored
- 11.1.2 Downloading ABAP2XLSX
- 11.1.3 Creating XLSX Files Using ABAP

### 11.2 Enhancing Custom Reports with ABAP2XLSX
- 11.2.1 Converting an ALV to an Excel Object
- 11.2.2 Changing Number and Text Formats
- 11.2.3 Establishing Printer Settings
- 11.2.4 Using Conditional Formatting
- 11.2.5 Creating Spreadsheets with Multiple Worksheets
- 11.2.6 Using Graphs and Pie Charts
- 11.2.7 Embedding Macros
- 11.2.8 Emailing the Result
- 11.2.9 Adding Hyperlinks to SAP Transactions

### 11.3 Tips and Tricks
- 11.3.1 Using the Enhancement Framework for Your Own Fixes
- 11.3.2 Creating a Reusable Custom Framework

### 11.4 Beyond Spreadsheets: Microsoft Word Documents
- 11.4.1 Installing the Tool
- 11.4.2 Creating a Template
- 11.4.3 Filling Out the Template Programmatically

### 11.5 Summary

## 12 Web Dynpro ABAP and Floorplan Manager

### 12.1 The Model-View-Controller Concept
- 12.1.1 Model
- 12.1.2 View
- 12.1.3 Controller

### 12.2 Building the WDA Application
- 12.2.1 Creating a Web Dynpro Component
- 12.2.2 Declaring Data Structures for the Controller
- 12.2.3 Establishing View Settings
- 12.2.4 Defining the Windows
- 12.2.5 Navigating between Views inside the Window
- 12.2.6 Enabling the Application to be Called
12.3 Coding the WDA Application .................................................. 715
12.3.1 Linking the Controller to the Model .................................. 715
12.3.2 Selecting Monster Records .................................................. 715
12.3.3 Navigating to the Single-Record View ................................ 721
12.4 Using Floorplan Manager to Create WDA Applications .......... 724
12.4.1 Creating an WDA Application using FPM Manually .......... 726
12.4.2 Creating a WDA Application Using FPM via BOPF integration .... 738
12.5 Unit Testing WDA Applications .............................................. 744
12.6 Making WDA Look Like SAP Fiori ........................................... 747
12.6.1 Enabling SAP Fiori in WDA via Classic Configuration ........ 747
12.6.2 Enabling SAP Fiori in WDA via New Customizing UI .......... 748
12.7 Touch Enablement of WDA Applications .................................. 752
12.8 Summary .................................................................................. 753

13 SAPUI5 ...................................................................................... 755

13.1 Basics .................................................................................... 757
13.1.1 What Is SAPUI5? ................................................................. 757
13.1.2 SAPUI5 versus SAP Fiori ..................................................... 758
13.2 Modern IDEs ........................................................................... 759
13.2.1 VS Code ............................................................................. 759
13.2.2 SAP Business Application Studio ........................................ 761
13.3 Creating an SAPUI5 Application Manually ......................... 763
13.3.1 VS Code ............................................................................. 764
13.3.2 SAP Business Application Studio ........................................ 787
13.4 Creating an SAPUI5 Application Automatically .................. 792
13.4.1 VS Code ............................................................................. 793
13.4.2 SAP Business Application Studio ........................................ 798
13.5 Extension Tools ....................................................................... 804
13.5.1 Guided Development ......................................................... 805
13.5.2 Adding Elements with OpenUI5 ......................................... 809
13.6 Importing SAPUI5 Applications into SAP ERP ................. 813
13.6.1 Storing the Application inside SAP ....................................... 814
13.6.2 Testing the SAPUI5 Application from within SAP ERP ...... 816
13.7 Unit Testing SAPUI5 Applications .......................................... 818
13.7.1 ESLint ................................................................................. 818
13.7.2 QUnit ................................................................................. 819

14 ABAP Channels .......................................................................... 827

14.1 General Concepts .................................................................. 828
14.1.1 ABAP Messaging Channels ................................................. 829
14.1.2 ABAP Push Channels ......................................................... 830
14.1.3 ABAP Daemons ................................................................. 831
14.2 ABAP Messaging Channels: SAP GUI Example ................... 833
14.2.1 Coding the Sending Application ......................................... 835
14.2.2 Coding the Receiving Application ....................................... 840
14.2.3 Watching the Applications Communicate ......................... 844
14.3 ABAP Push Channels: SAPUI5 Example ............................... 847
14.3.1 Coding the Receiving (Backend) Components .................... 848
14.3.2 Coding the Sending (Frontend) Application ....................... 856
14.4 Internet of Things Relevance .................................................. 857
14.5 Summary ................................................................................ 859

Conclusion ................................................................................... 861
The Author .................................................................................. 863
Index ............................................................................................ 865
Index

A

ABAP
ABAP 1809 ................................................. 301, 349
ABAP 1909 ................................................ 345
ABAP 7.3 ................................................. 269
ABAP 7.4 ................................................ 300, 309
ABAP 7.5 ................................................ 49, 61, 312, 317
ABAP 7.55 .................................. 145, 301–302, 306
development system ........................................ 39
event mechanism ........................................... 599
packages ..................................................... 41
quick assist ................................................. 49
ABAP Authority Check Test Helper API .......... 280
ABAP CDS Language Support ....................... 95
ABAP channel ............................................. 827
general concept .......................................... 828
ABAP Code Insight ........................................ 83
ABAP Console .......................................... 63
ABAP Continuous Integration plug-in ............ 80, 82
ABAP Coverage View ..................................... 271
ABAP Development Tools (ADT) ................. 32, 479
ABAP Doc .................................................. 62–63
ABAP for Key Users ................................... 196
ABAP in Eclipse ........................................... 32
ADT .......................................................... 32
connection .................................................. 38
missing method .......................................... 46
ABAP language version ................................ 146, 195
ABAP Managed Database Procedures (ADMP) ........................................ 292, 324, 329, 350
ABAP program ......................................... 355
CDS entities .................................................. 356
Eclipse ...................................................... 350–352
method definition ......................................... 367
ABAP messaging channel ............................ 828–829, 833
activity scope ............................................. 837
coding the receiving application .................... 840
coding the sending application ....................... 835
configuration ............................................. 836
example ..................................................... 844
framework ................................................ 838
receiver object ............................................ 842
SAP GUI .................................................... 833
send a message ............................................ 837
subscriber object .......................................... 843
user-specific settings .................................... 835
warning ..................................................... 835

ABAP Objects ............................................ 45
ABAP programming model for
SAP S/4HANA ............................................. 323
ABAP push channel ..................................... 828, 830, 848
code for incoming messages ......................... 850
coding receiving components ....................... 848
coding the sending application ..................... 856
configuration ............................................. 848
ON_MESSAGE method ................................ 851
SAPUI5 ..................................................... 847
testing the APC service ................................ 854
ABAP RESTful application programming
model ....................................................... 332, 341, 363
business logic ............................................. 453
JSON configuration file ................................. 562
preview SAP Fiori Elements ......................... 557
service binding .......................................... 556–557
service definition ....................................... 555
service layer .............................................. 554
tracking changes ........................................ 518
trigger MAIN method .................................. 564
unit tests ................................................... 559
versus BOPF ............................................... 453
ABAP Snippets ........................................... 95
ABAP SQL .................................................. 292–293, 331
new commands ......................................... 294
query ....................................................... 295
retrieve single record .................................... 298
ABAP SQL Test Double Framework ............... 321
ABAP Syntax Highlighting .............................. 95
ABAP Test Cockpit ...................................... 63, 82, 85, 359–360, 818
ABAP Test Cockpit checks ......................... 197
ABAP Test Double Framework (ATDF) ........... 269, 274, 277
GIVEN method .......................................... 276
THEN method ........................................... 277
verifying results .......................................... 275
WHEN method ........................................... 276
ABAP Unit ................................................. 233
ABAP Unit Authority Check Framework
API .......................................................... 282
ABAP Unit framework ................................ 818
ABAP Unit Runner ....................................... 271
ABAP Unit Test Double Framework ............... 820
ABAP Workbench ...................................... 31–32, 42, 48, 71, 538
ABAP_TRUE .............................................. 161
ABAP2DOCX ............................................. 687
invoice example ......................................... 687
type definitions .......................................... 682
<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse (Cont.)</td>
<td>37</td>
</tr>
<tr>
<td>welcome page</td>
<td>37</td>
</tr>
<tr>
<td>window layout</td>
<td>42</td>
</tr>
<tr>
<td>word wrap</td>
<td>67</td>
</tr>
<tr>
<td>Eiffel</td>
<td>226, 228</td>
</tr>
<tr>
<td>Ellison, Larry</td>
<td>291</td>
</tr>
<tr>
<td>ELSE clause</td>
<td>330</td>
</tr>
<tr>
<td>Embedded views</td>
<td>510</td>
</tr>
<tr>
<td>Entities</td>
<td>522, 526</td>
</tr>
<tr>
<td>Entity Manipulation Language (EML)</td>
<td>505, 508, 515, 519</td>
</tr>
<tr>
<td>Entity set</td>
<td>526</td>
</tr>
<tr>
<td>Enumeration</td>
<td>152, 154</td>
</tr>
<tr>
<td>Error function</td>
<td>786</td>
</tr>
<tr>
<td>Error handling</td>
<td>214, 220, 480, 485, 537</td>
</tr>
<tr>
<td>method</td>
<td>599</td>
</tr>
<tr>
<td>RESUME</td>
<td>222</td>
</tr>
<tr>
<td>RETRY</td>
<td>221</td>
</tr>
<tr>
<td>Error message</td>
<td>211</td>
</tr>
<tr>
<td>Errors</td>
<td>787</td>
</tr>
<tr>
<td>ELSInt</td>
<td>818</td>
</tr>
<tr>
<td>ETag</td>
<td>501</td>
</tr>
<tr>
<td>Excel</td>
<td>631</td>
</tr>
<tr>
<td>add object attributes</td>
<td>636</td>
</tr>
<tr>
<td>and ABAP</td>
<td>655</td>
</tr>
<tr>
<td>and XML</td>
<td>637</td>
</tr>
<tr>
<td>change numbers and text formats</td>
<td>641</td>
</tr>
<tr>
<td>create object</td>
<td>635</td>
</tr>
<tr>
<td>download spreadsheet to frontend</td>
<td>637</td>
</tr>
<tr>
<td>spreadsheet cell style</td>
<td>642</td>
</tr>
<tr>
<td>EXCEL_WRITER</td>
<td>637</td>
</tr>
<tr>
<td>Exception</td>
<td>61, 199, 201, 549</td>
</tr>
<tr>
<td>cleanup</td>
<td>219</td>
</tr>
<tr>
<td>examples</td>
<td>199</td>
</tr>
<tr>
<td>handling</td>
<td>201</td>
</tr>
<tr>
<td>method clean up</td>
<td>220</td>
</tr>
<tr>
<td>object</td>
<td>201, 214</td>
</tr>
<tr>
<td>raising</td>
<td>201, 213</td>
</tr>
<tr>
<td>recommended reading</td>
<td>232</td>
</tr>
<tr>
<td>text</td>
<td>211</td>
</tr>
<tr>
<td>Exception class</td>
<td>61, 199, 201–202, 417, 549</td>
</tr>
<tr>
<td>choosing type</td>
<td>206</td>
</tr>
<tr>
<td>constructor</td>
<td>208</td>
</tr>
<tr>
<td>creation</td>
<td>207</td>
</tr>
<tr>
<td>custom attributes</td>
<td>208</td>
</tr>
<tr>
<td>declaring</td>
<td>212</td>
</tr>
<tr>
<td>design</td>
<td>207</td>
</tr>
<tr>
<td>message classes</td>
<td>209</td>
</tr>
<tr>
<td>text</td>
<td>210</td>
</tr>
<tr>
<td>types</td>
<td>201</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>406, 412, 423</td>
</tr>
<tr>
<td>Existence check</td>
<td>309</td>
</tr>
<tr>
<td>Export parameter</td>
<td>226</td>
</tr>
<tr>
<td>EXPORTING parameter</td>
<td>42</td>
</tr>
<tr>
<td>Extended syntax check</td>
<td>55</td>
</tr>
<tr>
<td>External breakpoint</td>
<td>545–547, 857</td>
</tr>
<tr>
<td>F</td>
<td>465</td>
</tr>
<tr>
<td>Facet</td>
<td>602</td>
</tr>
<tr>
<td>Factory class</td>
<td>57, 249, 386</td>
</tr>
<tr>
<td>Factory method</td>
<td>57</td>
</tr>
<tr>
<td>return instance</td>
<td>250</td>
</tr>
<tr>
<td>method</td>
<td>250</td>
</tr>
<tr>
<td>Fast ALV Grid Object (FALV)</td>
<td>628</td>
</tr>
<tr>
<td>Favorites list</td>
<td>78</td>
</tr>
<tr>
<td>Feature control</td>
<td>469, 513</td>
</tr>
<tr>
<td>dynamic</td>
<td>514</td>
</tr>
<tr>
<td>Feeder</td>
<td>728</td>
</tr>
<tr>
<td>code</td>
<td>728</td>
</tr>
<tr>
<td>methods</td>
<td>729</td>
</tr>
<tr>
<td>Field catalog</td>
<td>595, 612</td>
</tr>
<tr>
<td>field symbol</td>
<td>507</td>
</tr>
<tr>
<td>FILTER</td>
<td>152, 177</td>
</tr>
<tr>
<td>Filter structure</td>
<td>389</td>
</tr>
<tr>
<td>FitNesse</td>
<td>259</td>
</tr>
<tr>
<td>Floorplan Manager (FFM)</td>
<td>31, 689, 724</td>
</tr>
<tr>
<td>BOM integration</td>
<td>738</td>
</tr>
<tr>
<td>drilldown</td>
<td>737</td>
</tr>
<tr>
<td>floorplans</td>
<td>725</td>
</tr>
<tr>
<td>GUI/BBS</td>
<td>727</td>
</tr>
<tr>
<td>guided activity floorplan</td>
<td>725</td>
</tr>
<tr>
<td>outlook</td>
<td>744</td>
</tr>
<tr>
<td>overview floorplan</td>
<td>725</td>
</tr>
<tr>
<td>quick activity floorplan</td>
<td>726</td>
</tr>
<tr>
<td>recommended reading</td>
<td>753</td>
</tr>
<tr>
<td>UI/BBS</td>
<td>727</td>
</tr>
<tr>
<td>FLUID tool</td>
<td>743</td>
</tr>
<tr>
<td>Font size</td>
<td>66</td>
</tr>
<tr>
<td>FOR</td>
<td>150</td>
</tr>
<tr>
<td>FOR ALL ENTRIES</td>
<td>186, 316</td>
</tr>
<tr>
<td>FOR/NEXT variations</td>
<td>181</td>
</tr>
<tr>
<td>Foreign keys</td>
<td>333</td>
</tr>
<tr>
<td>Forked repository</td>
<td>131</td>
</tr>
<tr>
<td>FORM routine</td>
<td>43, 45</td>
</tr>
<tr>
<td>Fragment</td>
<td>769</td>
</tr>
<tr>
<td>Friends</td>
<td>249</td>
</tr>
<tr>
<td>Function module</td>
<td>213, 215–216, 604</td>
</tr>
<tr>
<td>signature</td>
<td>215</td>
</tr>
<tr>
<td>Functional methods</td>
<td>351</td>
</tr>
<tr>
<td>G</td>
<td>449</td>
</tr>
<tr>
<td>Generic method</td>
<td>449</td>
</tr>
<tr>
<td>Generic user interface building blocks</td>
<td>727, 738</td>
</tr>
<tr>
<td>(GUI/BBS)</td>
<td>734</td>
</tr>
<tr>
<td>components</td>
<td>734</td>
</tr>
<tr>
<td>GET_ENTITYSET</td>
<td>538</td>
</tr>
<tr>
<td>query method</td>
<td>546</td>
</tr>
<tr>
<td>testing</td>
<td>544</td>
</tr>
<tr>
<td>Gherkin</td>
<td>821</td>
</tr>
<tr>
<td>Git</td>
<td>100</td>
</tr>
<tr>
<td>GitHub</td>
<td>100, 105, 136</td>
</tr>
<tr>
<td>account</td>
<td>101</td>
</tr>
<tr>
<td>create project</td>
<td>112</td>
</tr>
<tr>
<td>errors</td>
<td>105</td>
</tr>
<tr>
<td>Issue tab</td>
<td>127</td>
</tr>
<tr>
<td>new repository</td>
<td>113, 119</td>
</tr>
<tr>
<td>submit issue</td>
<td>127</td>
</tr>
<tr>
<td>GIVEN method</td>
<td>746</td>
</tr>
<tr>
<td>GIVEN_WHEN_THEN</td>
<td>322</td>
</tr>
<tr>
<td>Global class</td>
<td>70</td>
</tr>
<tr>
<td>Global temporary table</td>
<td>317–318</td>
</tr>
<tr>
<td>God class</td>
<td>571</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>181, 185</td>
</tr>
<tr>
<td>GUID</td>
<td>375, 396</td>
</tr>
<tr>
<td>key</td>
<td>387</td>
</tr>
<tr>
<td>Guided development</td>
<td>805</td>
</tr>
<tr>
<td>column property</td>
<td>806</td>
</tr>
<tr>
<td>generated code</td>
<td>808</td>
</tr>
<tr>
<td>overview</td>
<td>806</td>
</tr>
<tr>
<td>H</td>
<td>783</td>
</tr>
<tr>
<td>Handler class</td>
<td>479</td>
</tr>
<tr>
<td>Hard-coded restrictions</td>
<td>331</td>
</tr>
<tr>
<td>Hashed key</td>
<td>186</td>
</tr>
<tr>
<td>Head First Design Patterns</td>
<td>245</td>
</tr>
<tr>
<td>Helper class</td>
<td>392</td>
</tr>
<tr>
<td>Helper method</td>
<td>259, 589</td>
</tr>
<tr>
<td>Helper variable</td>
<td>171</td>
</tr>
<tr>
<td>Hollywood Principle</td>
<td>829</td>
</tr>
<tr>
<td>Hotspot</td>
<td>581</td>
</tr>
<tr>
<td>HTTP authentication</td>
<td>108</td>
</tr>
<tr>
<td>importing parameter</td>
<td>479, 486</td>
</tr>
<tr>
<td>Importing parameter</td>
<td>356</td>
</tr>
<tr>
<td>Importing table</td>
<td>578</td>
</tr>
<tr>
<td>Inbound plug</td>
<td>712, 722</td>
</tr>
<tr>
<td>Index</td>
<td>394</td>
</tr>
<tr>
<td>information/warning/error</td>
<td>216</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>582, 605, 611</td>
</tr>
<tr>
<td>Injection</td>
<td>242, 246, 248</td>
</tr>
<tr>
<td>injector</td>
<td>250</td>
</tr>
<tr>
<td>class</td>
<td>250</td>
</tr>
<tr>
<td>Inline declarations</td>
<td>64</td>
</tr>
<tr>
<td>INNER JOIN</td>
<td>310</td>
</tr>
<tr>
<td>Integrated data access</td>
<td>621</td>
</tr>
<tr>
<td>advance</td>
<td>623</td>
</tr>
<tr>
<td>calculated fields</td>
<td>625</td>
</tr>
<tr>
<td>class</td>
<td>625</td>
</tr>
<tr>
<td>selection criteria</td>
<td>622</td>
</tr>
<tr>
<td>Integrated development environment</td>
<td>31</td>
</tr>
<tr>
<td>Integration test</td>
<td>820</td>
</tr>
<tr>
<td>Interaction phase</td>
<td>476</td>
</tr>
<tr>
<td>Interface CDS entities</td>
<td>460</td>
</tr>
<tr>
<td>Interface view</td>
<td>326</td>
</tr>
<tr>
<td>Interfaces</td>
<td>192</td>
</tr>
<tr>
<td>Internal table</td>
<td>167, 396</td>
</tr>
<tr>
<td>deep structures</td>
<td>172</td>
</tr>
<tr>
<td>extracting</td>
<td>185</td>
</tr>
<tr>
<td>grouping</td>
<td>181</td>
</tr>
<tr>
<td>JOIN</td>
<td>319</td>
</tr>
<tr>
<td>new functions</td>
<td>177</td>
</tr>
<tr>
<td>queries</td>
<td>179</td>
</tr>
<tr>
<td>read</td>
<td>169</td>
</tr>
<tr>
<td>virtual sorting</td>
<td>187</td>
</tr>
<tr>
<td>Internet of Things (IoT)</td>
<td>828, 857</td>
</tr>
<tr>
<td>IS INITIAL statement</td>
<td>315</td>
</tr>
<tr>
<td>Isolation policy</td>
<td>829</td>
</tr>
<tr>
<td>J</td>
<td>783</td>
</tr>
<tr>
<td>java</td>
<td>31, 148, 755</td>
</tr>
<tr>
<td>javascript</td>
<td>755</td>
</tr>
<tr>
<td>library</td>
<td>757, 809</td>
</tr>
<tr>
<td>Joe Dolce principle</td>
<td>274</td>
</tr>
<tr>
<td>Join condition</td>
<td>459</td>
</tr>
<tr>
<td>JSon Voorhees</td>
<td>837</td>
</tr>
<tr>
<td>K</td>
<td>526</td>
</tr>
<tr>
<td>Key fields</td>
<td>526</td>
</tr>
<tr>
<td>L</td>
<td>304</td>
</tr>
<tr>
<td>LAG statement</td>
<td>304</td>
</tr>
<tr>
<td>Layout data property</td>
<td>706</td>
</tr>
</tbody>
</table>
Paul Hardy is a senior ABAP developer at Hanson and has worked on SAP rollouts at multiple companies all over the world. He joined Heidelberg Cement in the UK in 1990 and, for the first seven years, worked as an accountant. In 1997, a global SAP rollout came along, he jumped on board and has never looked back since. He has worked on country-specific SAP implementations in the United Kingdom, Germany, Israel, and Australia.

After starting off as a business analyst configuring the good old IMG, Paul swiftly moved on to the wonderful world of ABAP programming. After the initial run of data conversion programs, ALV reports, interactive Dynpro screens, and (urrggh) SAPscript forms, he yearned for something more and since then has been eagerly investigating each new technology as it comes out. Particular areas of interest in SAP are business workflow, B2B procurement (both point-to-point and SAP Ariba-based), logistics execution, and variant configuration, along with virtually anything new that comes along.

Paul can regularly be found blogging away on SAP Community and presenting at SAP conferences in Australia (Mastering SAP Technology and the SAP Australian User Group annual conference). If you happen to ever be at one of these conferences, Paul invites you to come and have a drink with him at the networking event in the evening and to ask him the most difficult questions you can think of (preferably SAP-related).