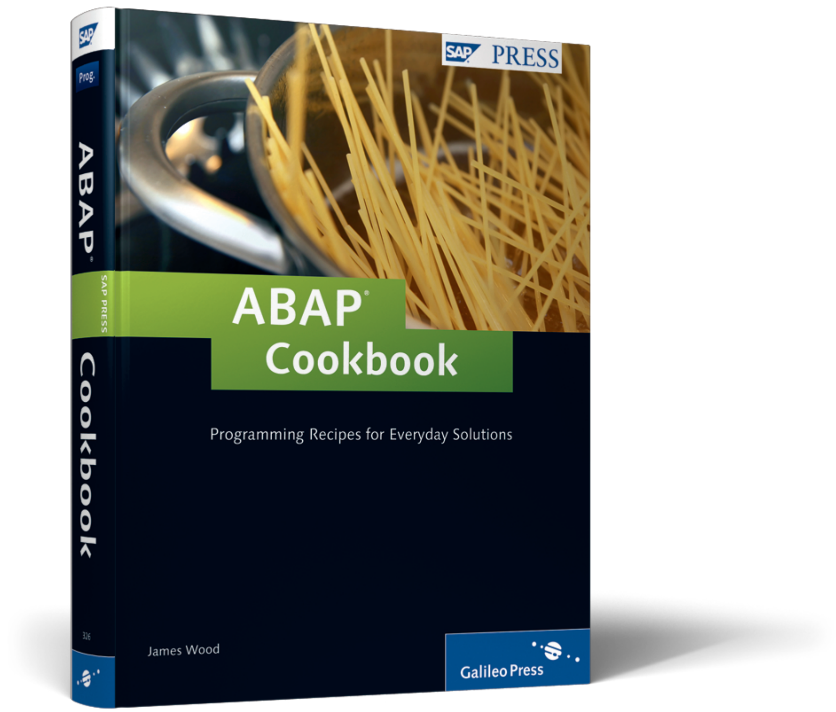
James Wood

ABAP™ Cookbook

Programming Recipes for Everyday Solutions



www.sap-press.com

Galileo Press

Bonn • Boston

Contents at a Glance

Contents

PART II Main Courses

Although amateur cooks may hesitate to experiment with spices, accomplished chefs know how to use them to create the perfect dish. As an ABAP developer, the same can be said of certain data types. In this chapter, we show you how you can use some of these types to improve the quality of your programs.

2 Working with Numbers, Dates, and Bytes

One of the nice things about working with an advanced programming language like ABAP is that you don't often have to worry about how that data is represented behind the scenes at the bits and bytes level; the language does such a good job of abstracting data that it becomes irrelevant. However, if you do come across a requirement that compels you to dig a little deeper, you'll find that ABAP also has excellent support for performing more advanced operations with elementary data types. In this chapter, we investigate some of these operations and show you techniques for using these features in your programs.

2.1 Numeric Operations

Whether it's keeping up with a loop index or calculating entries in a balance sheet, almost every ABAP program works with numbers on some level. Typically, whenever we perform operations on these numbers, we use basic arithmetic operators such as the `+` (addition), `-` (subtraction), `*` (multiplication), or `/` (division) operators. Occasionally, we might use the `MOD` operator to calculate the remainder of an integer division operation, or the `**` operator to calculate the value of a number raised to the power of another. However, sometimes we need to perform more advanced calculations. If you're a mathematics guru, then perhaps you could come up with an algorithm to perform these advanced calculations using the basic arithmetic operators available in ABAP. For the rest of us mere mortals, ABAP provides an extensive set of mathematics tools that can be used to simplify these requirements. In the next two sections, we'll examine these tools and see how to use them in your programs.

2.1.1 ABAP Math Functions

ABAP provides many built-in math functions that you can use to develop advanced mathematical formulas as listed in Table 2.1. In many cases, these functions can be called using any of the built-in numeric data types in ABAP (e.g., the `I`, `F`, and `P` data types). However, some of these functions require the precision of the floating point data type (see Table 2.1 for more details). Because ABAP supports implicit type conversion between numeric types, you can easily cast non-floating point types into floating point types for use within these functions.

Function	Supported Numeric Types	Description
`abs`	`(All)`	Calculates the absolute value of the provided argument.
`sign`	`(All)`	Determines the sign of the provided argument. If the sign is positive, the function returns 1; if it's negative, it returns -1; otherwise, it returns 0.
`ceil`	`(All)`	Calculates the smallest integer value that isn't smaller than the argument.
`floor`	`(All)`	Calculates the largest integer value that isn't larger than the argument.
`trunc`	`(All)`	Returns the integer part of the argument.
`frac`	`(All)`	Returns the fractional part of the argument.
`cos, sin, tan`	`F`	Implements the basic trigonometric functions.
`acos, asin, atan`	`F`	Implements the inverse trigonometric functions.
`cosh, sinh, tanh`	`F`	Implements the hyperbolic trigonometric functions.
`exp`	`F`	Implements the exponential function with a base $e \approx 2.7182818285$.
`log`	`F`	Implements the natural logarithm function.
`log10`	`F`	Calculates a logarithm using base 10.
`sqrt`	`F`	Calculates the square root of a number.

Table 2.1 ABAP Math Functions

The report program ZMATHDEMO shown in Listing 2.1 contains examples of how to call the math functions listed in Table 2.1 in an ABAP program. The output of this program is displayed in Figure 2.1.

```
REPORT zmathdemo.

START-OF-SELECTION.
CONSTANTS: CO_PI TYPE f VALUE '3.14159265'.
DATA: lv_result TYPE p DECIMALS 2.

lv_result = abs( -3 ).
WRITE: / 'Absolute Value:      ', lv_result.

lv_result = sign( -12 ).
WRITE: / 'Sign:                ', lv_result.

lv_result = ceil( '4.7' ).
WRITE: / 'Ceiling:             ', lv_result.

lv_result = floor( '4.7' ).
WRITE: / 'Floor:               ', lv_result.

lv_result = trunc( '4.7' ).
WRITE: / 'Integer Part:        ', lv_result.

lv_result = frac( '4.7' ).
WRITE: / 'Fractional Part:     ', lv_result.

lv_result = sin( CO_PI ).
WRITE: / 'Sine of PI:          ', lv_result.

lv_result = cos( CO_PI ).
WRITE: / 'Cosine of PI:        ', lv_result.

lv_result = tan( CO_PI ).
WRITE: / 'Tangent of PI:       ', lv_result.

lv_result = exp( '2.3026' ).
WRITE: / 'Exponential Function:', lv_result.

lv_result = log( lv_result ).
WRITE: / 'Natural Logarithm:   ', lv_result.
```

```
lv_result = log10( '1000.0' ).
WRITE: / 'Log Base 10 of 1000: ', lv_result.

lv_result = log( 8 ) / log( 2 ).
WRITE: / 'Log Base 2 of 8:     ', lv_result.

lv_result = sqrt( '16.0' ).
WRITE: / 'Square Root:         ', lv_result.
```

Listing 2.1 Working with ABAP Math Functions

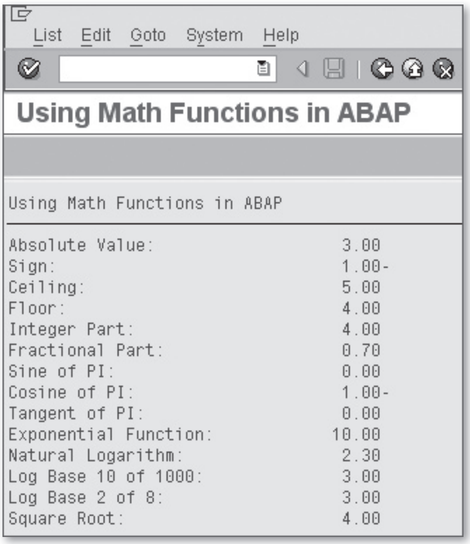


Figure 2.1 Output Generated by Report ZMATHDEMO

The values of the function calls can be used as operands in more complex expressions. For example, in Listing 2.1, notice how we're calculating the value of `log( 8 )`. Here, we use the change of base formula `log( x ) / log( b )` (where `b` refers to the target base, and `x` refers to the value applied to the logarithm function) to derive the base 2 value. Collectively, these functions can be combined with typical math operators to devise some very complex mathematical formulas.

2.1.2 Generating Random Numbers

Computers live in a logical world where everything is supposed to make sense. Whereas this characteristic makes computers very good at automating many kinds of tasks, it can also make it somewhat difficult to model certain real-world phenomena. Often, we need to simulate *imperfection* in some form or another. One common method for achieving this is to produce randomized data using random number generators. Random numbers are commonly used in statistics, cryptography, and many kinds of scientific applications. They are also used in algorithm design to implement *fairness* and to simulate useful metaphors applied to the study of artificial intelligence (e.g., genetic algorithms with randomized mutations, etc.).

SAP provides random number generators for all of the built-in numeric data types via a series of ABAP Objects classes. These classes begin with the prefix `CL_ABAP_RANDOM` (e.g., `CL_ABAP_RANDOM_FLOAT`, `CL_ABAP_RANDOM_INT`, etc.). Though none of these classes inherit from the `CL_ABAP_RANDOM` base class, they do use its features behind the scenes using a common OO technique called *composition*. Composition basically implies that one class delegates certain functionality to an instance of another class. The UML class diagram shown in Figure 2.2 shows the basic structure of the provided random number generator classes.

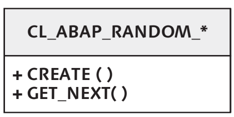


Figure 2.2 Basic UML Class Diagram for Random Number Generators

Unlike most classes where you create an object using the `CREATE OBJECT` statement, instances of random number generators must be created via a call to a factory class method called `CREATE()`. The signature of the `CREATE()` method is shown in Figure 2.3. Here, you can see that the method defines an importing parameter called `SEED` that *seeds* the pseudo-random number generator algorithm that is used behind the scenes to generate the random numbers. In a pseudo-random number generator, random numbers are generated in sequence based on some calculation performed using the seed. Thus, a given seed value causes the random number generator to generate the same sequence of random numbers each time.

The `CREATE()` method for class `CL_ABAP_RANDOM_INT` also provides `MIN` and `MAX` parameters that can place limits around the random numbers that are generated (e.g., a range of 1-100, etc.). The returning `PRNG` parameter represents the generated random number generator instance. Once created, you can begin retrieving random numbers via a call to the `GET_NEXT()` instance method.

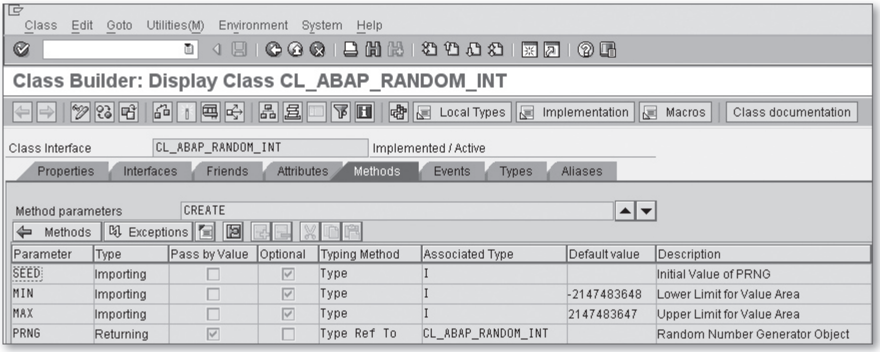


Figure 2.3 Signature of Class Method CREATE()

To demonstrate how these random number generator classes work, let's consider an example program. Listing 2.2 contains a simple report program named `ZSCRAMBLER` that defines a local class called `LCL_SCRAMBLER`. The `LCL_SCRAMBLER` class includes an instance method `SCRAMBLE()` that can be used to randomly scramble around the characters in a string. This primitive implementation creates a random number generator to produce random numbers in the range of [`0...{String Length}`]. Perhaps the most complex part of the implementation is related to the fact that random number generators produce some duplicates along the way. Therefore, we have to make sure that we haven't used the randomly generated number previously to make sure that each character in the original string is copied into the new one.

```
REPORT zscrambler.

CLASS lcl_scrambler DEFINITION.
  PUBLIC SECTION.
    METHODS: scramble IMPORTING im_value TYPE clike
                      RETURNING VALUE(re_svalue) TYPE string
                      EXCEPTIONS cx_abap_random.

  PRIVATE SECTION.
    CONSTANTS: CO_SEED TYPE i VALUE 100.

    TYPES: BEGIN OF ty_index,
             index TYPE i,
           END OF ty_index.
ENDCLASS.

CLASS lcl_scrambler IMPLEMENTATION.
  METHOD scramble.
```

```
*   Method-Local Data Declarations:
    DATA: lv_length  TYPE i,
          lv_min     TYPE i VALUE 0,
          lv_max     TYPE i,
          lo_prng    TYPE REF TO cl_abap_random_int,
          lv_index   TYPE i,
          lt_indexes TYPE STANDARD TABLE OF ty_index.
    FIELD-SYMBOLS:
      <lfs_index> LIKE LINE OF lt_indexes.

*   Determine the length of the string as this sets the
*   bounds on the scramble routine:
    lv_length = strlen( im_value ).
    lv_max = lv_length - 1.

*   Create a random number generator to return random
*   numbers in the range of 1..{String Length}:
    CALL METHOD cl_abap_random_int=>create
      EXPORTING
        seed   = CO_SEED
        min    = lv_min
        max    = lv_max
      RECEIVING
        prng   = lo_prng.

*   Add the characters from the string in random order to
*   the result string:
    WHILE strlen( re_svalue ) LT lv_length.
      lv_index = lo_prng->get_next( ).
      READ TABLE lt_indexes TRANSPORTING NO FIELDS
            WITH KEY index = lv_index.
      IF sy-subrc EQ 0.
        CONTINUE.
      ENDIF.

      CONCATENATE re_svalue im_value+lv_index(1)
             INTO re_svalue.
      APPEND INITIAL LINE TO lt_indexes
             ASSIGNING <lfs_index>.
      <lfs_index>-index = lv_index.
    ENDWHILE.
  ENDMETHOD.
ENDCLASS.
```

```
START-OF-SELECTION.
* Local Data Declarations:
  DATA: lo_scrambler TYPE REF TO lcl_scrambler,
        lv_scrambled TYPE string.

* Use the scrambler to scramble around a word:
  CREATE OBJECT lo_scrambler.
  lv_scrambled = lo_scrambler->scramble( 'Andersen' ).
  WRITE: / lv_scrambled.
```

Listing 2.2 Using Random Number Generators in ABAP

Obviously, a simple scrambler routine like the one shown in Listing 2.2 isn't production quality. Nevertheless, it does give you a glimpse of how you can use random number generators to implement some interesting algorithms. As a reader exercise, you might think about how you could use random number generators to implement an `UNSCRAMBLE()` method to unscramble strings generated from calls to method `SCRAMBLE()`.

2.2 Date and Time Processing

Online transaction processing (OLTP) systems such as the ones that make up the SAP Business Suite maintain quite a bit of time-sensitive data, so it's important that you understand how to work with the built-in date and time types provided in ABAP. In the following subsections, we discuss these types and explain how to use them to perform calculations and conversions.

2.2.1 Understanding ABAP Date and Time Types

ABAP provides two built-in types to work with dates and times: the `D` (date) data type and the `T` (time) data type. Both of these types are fixed-length character types that have the form `YYYYMMDD` and `HHMMSS`, respectively. In addition to these built-in types, the ABAP Dictionary types `TIMESTAMP` and `TIMESTAMPL` are being used more and more in many standard application tables, and so on, to store a timestamp in the UTC format.[1] Table 2.2 shows the basic date and time types available in ABAP.

1 The term "UTC" is an abbreviation for "Consolidated Universal Time," which is a time standard based on the International Atomic Time standard. UTC is roughly equivalent to the Greenwich Mean Time standard (or GMT) which refers to the mean solar time at the Royal Observatory in Greenwich, London. Collectively, these standards define a global time standard that can be used to convert a given time to local time, and vice versa.

Data Type	Description
D	A built-in fixed-length date type of the form YYYYMMDD. For example, the value 20100913 represents the date September 13, 2010.
T	A built-in fixed-length time type of the form HHMMSS. For example, the value 102305 represents the time 10:23:05 AM.
TIMESTAMP (Type P – Length 8 No decimals)	An ABAP Dictionary type used to represent short timestamps in the form YYYYMMDDhhmmss. For example, the value 20100913102305 represents the date September 13, 2010 at 10:23:05 AM.
TIMESTAMPL (Type P - Length 11 Decimals 7)	An ABAP Dictionary type used to represent long timestamps in the form YYYYMMDDhhmmssmmmuuun. The additional digits mmmuuun represent fractions of a second.

Table 2.2 ABAP Date and Time Data Types

2.2.2 Date and Time Calculations

When you're working with dates, you often need to perform various calculations to compute the difference between two dates, make comparisons, or determine a valid date range. As we mentioned in Section 2.2.1, Understanding ABAP Date and Time Types, the built-in date and time types in ABAP are *character types*, not numeric types. Nevertheless, the ABAP runtime environment allows you to perform basic numeric operations on these types by implicitly converting them to numeric types behind the scenes.

The code excerpt shown in Listing 2.3 demonstrates how these calculations work. Initially, the variable lv_date is assigned the value of the current system date (e.g., the system field SY-DATUM). Next, we increment that date value by 30. In terms of a date calculation in ABAP, this implies that we're increasing the *day* component of the date object by 30 days. Here, note that the ABAP runtime environment is smart enough to *roll over* the date value whenever it reaches the end of a month, and so on. In other words, you can rely on the system to ensure that you don't calculate an invalid date value (e.g., 01/43/2011).

```
DATA: lv_date TYPE d.
lv_date = sy-datum.
WRITE: / 'Current Date:', lv_date MM/DD/YYYY.
```

```
lv_date = lv_date + 30.
WRITE: / 'Future Date:', lv_date MM/DD/YYYY.
```

Listing 2.3 Performing Date Calculations in ABAP

Time calculations in ABAP work very similarly to the date calculations shown in Listing 2.3. With time calculations, the computation is based upon the *seconds* component of the time object. The code in Listing 2.4 shows how we can increment the current system time by 90 seconds using basic time arithmetic.

```
DATA: lv_time TYPE t.
lv_time = sy-uzeit.
WRITE /(60) lv_time USING EDIT MASK
  'The current time is __:__:__'.
lv_time = lv_time + 90.
WRITE /(60) lv_time USING EDIT MASK
  'A minute and a half from now it will be __:__:__'.
```

Listing 2.4 Performing Time Calculations in ABAP

In addition to typical numeric calculations, you also have the option of working with date/time fields using normal character-based semantics. For instance, you can use the offset/length functionality to initialize date or time components. The code excerpt in Listing 2.5 demonstrates how you can adjust the date 02/13/2003 to 01/13/2003 using offset/length semantics.

```
DATA: lv_date TYPE d VALUE '20030213'.
WRITE: / lv_date MM/DD/YYYY.
lv_date+4(2) = '01'.
WRITE: / lv_date MM/DD/YYYY.
```

Listing 2.5 Manipulating a Date Using Offset/Length Functionality

2.2.3 Working with Timestamps

If you've been working with some of the newer releases of the products in the SAP Business Suite, you may have encountered certain applications that use the `TIMESTAMP` or `TIMESTAMPL` data types to store time-sensitive data. As you can see in Table 2.2, these ABAP Dictionary types store timestamps with varying degrees of accuracy. Interestingly, though these types aren't built-in types like `D` or `T`, ABAP does provide some native support for them in the form of a couple of built-in statements. In addition, SAP also provides a system class called `CL_ABAP_TSTMP`, which can be used to simplify the process of working with timestamps. We investigate these features in the following subsections.

Retrieving the Current Timestamp

You can retrieve the current system time and store it in a timestamp variable using the `GET TIME STAMP` statement whose syntax is demonstrated in Listing 2.6. The `GET TIME STAMP` statement stores the timestamp in a shorthand or longhand format depending upon the type of the timestamp data object used after the `FIELD` addition. The timestamp value is encoded using the UTC standard.

```
DATA: lv_tstamp_s TYPE timestamp,
      lv_tstamp_l TYPE timestampl.
GET TIME STAMP FIELD lv_tstamp_s.
WRITE: / 'Short Time Stamp:', lv_tstamp_s
  TIME ZONE sy-zonlo.
GET TIME STAMP FIELD lv_tstamp_l.
WRITE: / 'Long Time Stamp: ', lv_tstamp_l
  TIME ZONE sy-zonlo.
```

Listing 2.6 Using the GET TIME STAMP Statement

Looking at the code excerpt in Listing 2.6, you can see that we're displaying the timestamp using the `TIME ZONE` addition of the `WRITE` statement. This addition formats the output of the timestamp according to the rules for the time zone specified. In Listing 2.6, we used the system field `SY-ZONLO` to display the *local time zone* configured in the user's preferences. However, we could have just as easily used a data object of type `TIMEZONE`, or even a hard-coded literal such as `'CST'`.

Time Zones

For a complete list of time zones configured in the system, have a look at the contents of ABAP Dictionary Table TTZZ.

Converting Timestamps

You can convert a timestamp to a date/time data object and vice versa using the `CONVERT` statement in ABAP. Listing 2.7 shows the syntax used to convert a timestamp into data objects of type `D` and `T`. The `TIME ZONE` addition adjusts the UTC date/time value within the timestamp in accordance with a particular time zone. Additionally, the optional `DAYLIGHT SAVING TIME` addition can be used to determine whether or not the timestamp value happens to coincide with daylight savings time. If it does, the `lv_dst` variable has the value `'X'`; otherwise, it's blank.

This feature can be helpful in differentiating between timestamp values that lie within the transitional period between *summer time* and *winter time*.[2]

```
CONVERT TIME STAMP lv_tstamp TIME ZONE lv_tzone
   INTO [ DATE lv_date ] [ TIME lv_time ]
   [DAYLIGHT SAVING TIME lv_dst].
```

Listing 2.7 Syntax of CONVERT TIME STAMP Statement

Listing 2.8 shows how the `CONVERT TIME STAMP` statement is used to convert the current system timestamp to date/time data objects using the local time zone.

```
TYPE-POOLS: abap.
DATA: lv_tstamp TYPE timestamp,
      lv_date   TYPE d,
      lv_time   TYPE t,
      lv_dst    TYPE abap_bool.

GET TIME STAMP FIELD lv_tstamp.
CONVERT TIME STAMP lv_tstamp TIME ZONE sy-zonlo
   INTO DATE lv_date TIME lv_time
   DAYLIGHT SAVING TIME lv_dst.

WRITE: / 'Today's date is:    ', lv_date MM/DD/YYYY.
WRITE: /(60) lv_time USING EDIT MASK
             'The current time is: __:__:__'.

IF lv_dst EQ abap_true.
  WRITE: / 'In daylight savings time...'.
ELSE.
  WRITE: / 'Not in daylight savings time...'.
ENDIF.
```

Listing 2.8 Converting Timestamps to Date/Time Objects

To create a timestamp using a date/time object, you can use the syntax variant of the `CONVERT` statement shown in Listing 2.9. The date/time values are qualified using the `TIME ZONE` addition so that the appropriate offsets can be applied as the UTC timestamp is generated.

2 For a complete list of daylight savings time rules, have a look at the contents of the ABAP Dictionary table TTZDV.

```
CONVERT DATE lv_date
       [TIME lv_time [DAYLIGHT SAVING TIME lv_dst]]
   INTO TIME STAMP lv_tstamp TIME ZONE lv_tzone.
```

Listing 2.9 Syntax of CONVERT DATE Statement

The code excerpt in Listing 2.10 shows how the `CONVERT DATE` statement can be used to generate a timestamp object from a date/time object.

```
TYPE-POOLS: abap.
DATA: lv_tstamp TYPE timestamp,
      lv_date   TYPE d,
      lv_time   TYPE t,
      lv_dst    TYPE abap_bool.

lv_date = sy-datum.
lv_time = sy-uzeit.

CONVERT DATE lv_date TIME lv_time
   INTO TIME STAMP lv_tstamp TIME ZONE sy-zonlo.

WRITE: / 'Time Stamp Value:', lv_tstamp TIME ZONE sy-zonlo.
```

Listing 2.10 Creating a Timestamp from a Date/Time Object

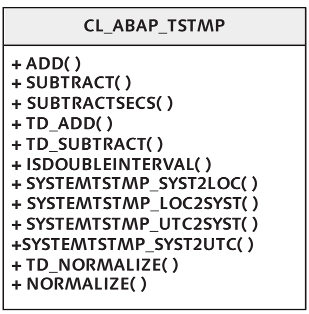


Figure 2.4 UML Class Diagram for Class CL_ABAP_TSTMP

Timestamp Operations Using System Class CL_ABAP_TSTMP

Unlike the native `D` and `T` types, the ABAP runtime environment doesn't have built-in functionality to perform calculations on timestamps (e.g., add or subtract, etc.). Instead, SAP provides a system class called `CL_ABAP_TSTMP` for this purpose. Figure 2.4 contains a UML class diagram that shows the publicly available methods provided in this class. As you would expect, there are various forms of `ADD()` and

SUBTRACT() methods to perform timestamp calculations. In addition, a series of conversion methods (e.g., SYSTEMTSTMP_SYST2LOC(), etc.) can be used to convert a timestamp to various time zones, a Boolean method called ISDOUBLEINTERVAL() can be used to determine if a timestamp is in daylight savings time, and a couple of methods can be used to *normalize* a timestamp. Here, normalization implies that an invalid time value such as 10:30:60 would be adjusted to the value 10:31:00.

In UML class diagram notation, methods that are underlined are defined as *class methods*. Class methods can be invoked without first creating an instance of the class in which they are defined, as evidenced in the code excerpt shown in Listing 2.11. Here, we're using the class method ADD() to add 75 seconds to the current system time.

```
DATA: lv_tstamp TYPE timestamp,
      lv_date   TYPE d,
      lv_time   TYPE t.

GET TIME STAMP FIELD lv_tstamp.
WRITE: / 'Time Stamp Value:', lv_tstamp TIME ZONE sy-zonlo.

TRY.
  CALL METHOD cl_abap_tstmp=>add
    EXPORTING
      tstmp   = lv_tstamp
      secs    = 75
    RECEIVING
      r_tstmp = lv_tstamp.
CATCH CX_PARAMETER_INVALID_RANGE.
CATCH CX_PARAMETER_INVALID_TYPE.
ENDTRY.

WRITE: / 'Time Stamp Value:', lv_tstamp TIME ZONE sy-zonlo.
```

Listing 2.11 Working with Timestamps Using CL_ABAP_TSTMP

The call signatures of most of the other methods in class CL_ABAP_TSTMP are similar to the ADD() method demonstrated in Listing 2.11. For more details concerning the functionality of particular methods in this class, see the class/method documentation for this class in the Class Builder (Transaction SE24).

2.2.4 Calendar Operations

So far, our discussion on dates has focused on raw calculations and conversions.

However, many typical use cases in the business world require that we look at dates from a semantic point of view. For example, you might ask whether or not the date 1/13/2010 is a working day, or whether 4/4/2010 is a holiday. The answers to these kinds of questions require the use of a *calendar*. Fortunately, SAP provides a very robust set of calendaring features straight out of the box with SAP NetWeaver AS ABAP.

The SAP Calendar is maintained in a client-specific manner inside the SAP Customizing implementation guide (Transaction SPRO). Depending on how your system is set up, you might have a project-specific implementation guide. However, for the purposes of this discussion, we assume that you're using the default SAP Reference Implementation Guide (IMG). You can access this guide by clicking on the button labeled SAP Reference IMG on the initial screen of Transaction SPRO (see Figure 2.5).

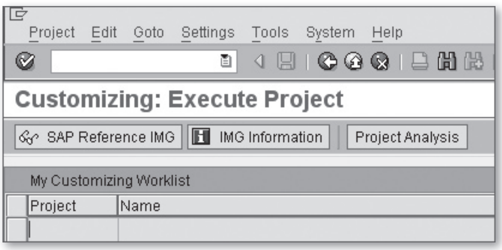


Figure 2.5 Initial Screen of Transaction SPRO

Inside the SAP Reference IMG, you can find the SAP Calendar under the navigation path SAP NetWeaver • General Settings • Maintain Calendar (see Figure 2.6).

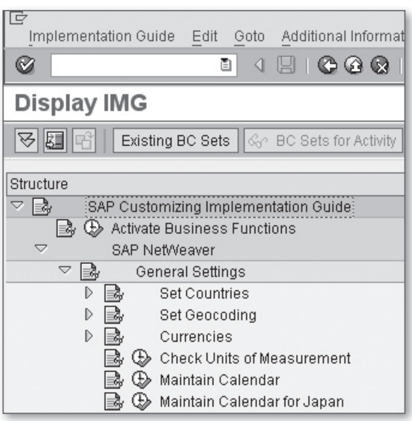


Figure 2.6 Navigating to the SAP Calendar in the IMG

Figure 2.7 shows the main menu of the SAP Calendar transaction. From here, you can configure subobjects such as public holidays, holiday calendars, and factory calendars. By default, an SAP NetWeaver system comes preconfigured with some typical settings in these subareas. However, you're also free to create customized holidays and calendars as needed.

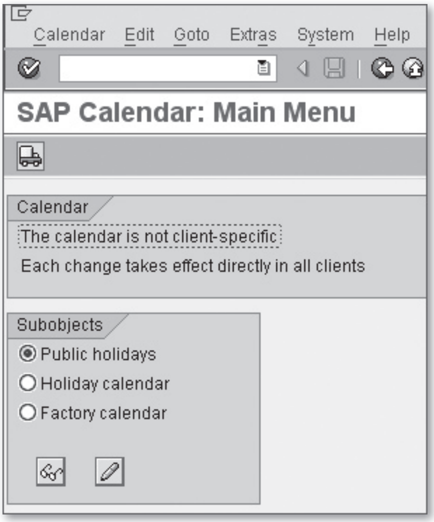


Figure 2.7 Maintaining the SAP Calendar in the IMG

After the SAP Calendar is configured properly, you can use this data to perform various types of calculations. Table 2.3 shows some useful function modules that leverage this data to determine whether or not a given date is a working day, holiday, and so on. You can find out more information about these function modules in the documentation provided for each module in the Function Builder (Transaction SE37).

Function Name	Description
DATE_COMPUTE_DAY	Computes the day of the week for a given date. Day values are calculated as 1 (Monday), 2 (Tuesday), and so on.
DATE_COMPUTE_DAY_ENHANCED	Computes the day of the week just like DATE_COMPUTE_DAY; also returns the day value as text (e.g., TUESDAY, etc.).

Table 2.3 Useful Date Functions in Function Group SCAL

Function Name	Description
DATE_CONVERT_TO_FACTORYDATE	Calculates the factory date value for a given date. Also provides an indicator that confirms whether or not the given date is considered a working day according to the selected factory calendar.
DATE_GET_WEEK	Determines the week of the year for the given date. For example, the date 9/13/2010 would be the 37th week of the year 2010.
FACTORYDATE_CONVERT_TO_DATE	Converts a factory date value back into a date object.
HOLIDAY_CHECK_AND_GET_INFO	Tests to determine whether or not a given date is a holiday based on the configured holiday calendar.
WEEK_GET_FIRST_DAY	Calculates the first day of a given week.

Table 2.3 Useful Date Functions in Function Group SCAL (Cont.)

2.3 Bits and Bytes

Modern programming languages do such a tremendous job of abstracting the complexities of computer architectures that, these days, we seldom have any need to work at the bits and bytes level. However, with the advent of Unicode, it's becoming more important to understand how to work at this level because many external data sources encode their data using multi-byte encodings — as opposed to the single-byte code pages normally used in ABAP (e.g., ASCII, etc.). In addition, knowledge of this area can be quite handy in other applications, as you'll see in a moment.

2.3.1 Introduction to the Hexadecimal Type in ABAP

Normally, whenever we talk about the built-in native data types provided in the ABAP programming language, we focus our attention around the numeric and character data types. However, ABAP also provides a hexadecimal data type (X) that is used to represent individual bytes in memory. The values stored in the individual bytes are represented as two-digit hexadecimal numbers.

Binary and Hexadecimal Numbers

If you have never worked with binary or hexadecimal numbers before, then a brief introduction is in order. A *byte* is a unit of measure for memory inside of a computer. Each byte is comprised of 8 bits. The term *bit* is an abbreviation for *binary digit*. A bit can have one of two logical values: `1` (or true) or `0` (or false). In terms of computer circuitry, bits that have the value `1` are turned *on*, while those that have the value `0` are turned *off*.

The binary (or base-2) number system represents numeric values using binary digits. Figure 2.8 shows an example of an 8-bit binary number whose decimal value is 170. As you can see, reading from right to left, the value of each bit is calculated by multiplying one or zero (i.e., the bit value) by two raised to the power of the current index (where indexes start at zero).

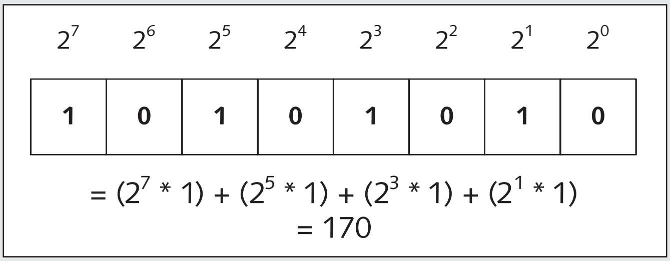


Figure 2.8 Example of an 8-Bit Binary Number

Binary numbers can be very difficult to work with if you're not a computer. Therefore, the values of bytes are often represented using the hexadecimal (or base-16) numbering system. Each hexadecimal digit is in the range [0123456789ABCDEF], where A = 10, B = 11, C = 12, and so on. Conveniently, each hexadecimal digit can hold any possible value of 4 bits (commonly called a *nibble*). Therefore, two hexadecimal digits can be used to represent a single byte of information in memory.

In addition to the fixed length `X` data type, ABAP also provides the `XSTRING` variable-length hexadecimal type, which is commonly used in various input/output (I/O) operations. Here, as is the case with the `C` and `STRING` data types described in Chapter 1, String Processing Techniques, there is a trade-off between performance and flexibility.

Now that you know a little bit about the hexadecimal type, let's take a look at the types of operations you can perform on data objects of this type. The following sections describe the built-in bitwise operators available in ABAP.

2.3.2 Reading and Writing Individual Bits

You can use the GET BIT and SET BIT statements to read and write individual bits of a hexadecimal data object. The general syntax of these statements is shown in Listing 2.12 and Listing 2.13, respectively.

```
GET BIT lv_index OF lv_hex INTO lv_bit.
```

Listing 2.12 Syntax of GET BIT Statement

```
SET BIT lv_index OF lv_hex TO lv_bit.
```

Listing 2.13 Syntax of SET BIT Statement

To demonstrate how these statements work, let's consider an example. Listing 2.14 contains a contrived piece of sample code that swaps the first byte of a two-byte hexadecimal data object with the last byte by manipulating individual bits internally. For good measure, we also shift the bits around one more time at the end of the code snippet, using the SHIFT statement in *byte mode*.

```
DATA: lv_hex(2)    TYPE x VALUE 'F00F',
      lv_front_idx TYPE i,
      lv_back_idx  TYPE i,
      lv_front_bit TYPE i,
      lv_back_bit  TYPE i.
WRITE: / lv_hex.
DO 8 TIMES.
  lv_front_idx = sy-index.
  lv_back_idx = lv_front_idx + 8.

  GET BIT lv_front_idx OF lv_hex INTO lv_front_bit.
  GET BIT lv_back_idx OF lv_hex INTO lv_back_bit.

  SET BIT lv_front_idx OF lv_hex TO lv_back_bit.
  SET BIT lv_back_idx OF lv_hex TO lv_front_bit.
ENDDO.
WRITE: / lv_hex.
SHIFT lv_hex BY 1 PLACES CIRCULAR IN BYTE MODE.
WRITE: / lv_hex.
```

Listing 2.14 Reading and Writing Bits in ABAP

In and of itself, low-level bit manipulation isn't all that exciting. However, there are situations where it can be quite useful.

For example, let's imagine you're working on a problem where you need to work with arbitrarily large numbers that exceed the limits of the built-in ABAP numeric types. One way other modern programming languages, such as Java or .NET, get around this limitation is by developing a so-called numeric *wrapper class*. For instance, the `java.math.BigInteger` class provided with the Java 2 SDK is used to represent arbitrarily large integer values. Internally, bitwise operators are used to mimic the behavior of a normal primitive type represented in two's complement notation.[3] Because this implementation is open source, it wouldn't be too difficult to reverse-engineer an ABAP version of this class to suit your purposes.

2.3.3 Bitwise Logical Operators

In addition to the `GET BIT` and `SET BIT` statements, ABAP also provides a series of bitwise logical operators that can be used to build Boolean algebraic expressions. If you aren't familiar with Boolean algebra, there are many excellent resources available online — simply search for the term "Boolean Algebra," and you'll find a wealth of information. Of course, even if you have worked with Boolean operators before, you might need a bit of a refresher. Table 2.4 depicts a *truth table* that shows the values generated when applying the Boolean `AND`, `OR`, or `XOR` operators against the two bit values contained in Field A and Field B.

Field A	Field B	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 2.4 Truth Table for Boolean Operators

Table 2.5 shows the bitwise operators provided with the ABAP language. Just like normal arithmetic operators, the bitwise operators can be combined in complex expressions using parentheses, and so on.

3 The two's complement notation is a common system used to represent signed integers in computers.

Bitwise Operator	Description
`BIT-NOT`	Unary operator that flips all of the bits in the hexadecimal number to the opposite value. For example, applying this operator to a hexadecimal number having the bit-level value `10101010` (e.g., `'AA'`) would yield `01010101`.
`BIT-AND`	Binary operator that compares each field bit-by-bit using the Boolean `AND` operator.
`BIT-XOR`	Binary operator that compares each field bit-by-bit using the Boolean `XOR` (or *eXclusive OR*) operator.
`BIT-OR`	Binary operator that compares each field bit-by-bit using the Boolean `OR` operator.

Table 2.5 Bitwise Logical Operators in ABAP

To see the power of bitwise operators such as the ones listed in Table 2.5, it's useful to consider an example. Imagine that you are tasked with building a custom document management system. One of the requirements of this system is to be able to assign rights permissions to the individual documents maintained in the system. For the purposes of this simple example, let's assume that the possible permissions are *Create*, *Remove*, *Update*, and *Display*.

One way to store these assignments might be to create a database table that contained a series of *flag* columns to indicate whether or not a user had a particular permission for a given document. Unfortunately, there are a couple of problems with this approach. First of all, it requires that we create separate fields for each possible permission type. As the system grows, additional permission types require a modification to the database table. This phenomenon leads into the second problem — namely, space. In other words, each additional flag column adds another byte or two of storage to every row in the table. Of course, another option is to capture the permissions in separate rows. Still, either way you slice it, this can get expensive from a storage perspective.

Instead of creating a new flag column each time we want to add a new permission type to our system, what if we could figure out a way to store a bunch of Boolean flags in a single field? Naturally, the hexadecimal data type lends itself well to this kind of storage operation because it can be used as a type of *bit mask* to represent a large number of flags at the bit level. For example, a single byte bit mask could represent up to 28, or 256, possible values, leaving us plenty of room to grow. The values of the individual Boolean flags can then be set using bitwise operators. Collectively, the process of representing a series of flags at the bit level and manipulating those flags using bitwise operators is referred to as *bit masking*.

The code excerpt in Listing 2.15 demonstrates how bit masking works using the ABAP bitwise logical operators. To keep things simple, we've created an interface that contains constants to represent the possible permission values (e.g., CO_CREATE, etc.). These permission values are assigned to a display-only user using the BIT-OR operator, which effectively works like an addition operator in this case. We can then confirm whether or not the user has a given permission by applying the BIT-AND operator. Here, the result matches the permission constant bit-for-bit if the particular permission has been assigned. This can be confirmed by using the equality operator in an IF statement. In the example, the user has *Display* permissions but not *Create* permissions.

```
INTERFACE lif_permissions.
  CONSTANTS: CO_CREATE  TYPE x VALUE '01',
             CO_REMOVE  TYPE x VALUE '02',
             CO_UPDATE  TYPE x VALUE '04',
             CO_DISPLAY TYPE x VALUE '08'.
ENDINTERFACE.

DATA: lv_display_user TYPE x,
      lv_permission   TYPE x.

* Assign read-only access to a display user:
lv_display_user =
  lv_display_user BIT-OR lif_permissions=>CO_DISPLAY.

* Check the user's permissions:
lv_permission =
  lv_display_user BIT-AND lif_permissions=>CO_DISPLAY.
IF lv_permission EQ lif_permissions=>CO_DISPLAY.
  WRITE: / 'User has display only access.'.
ELSE.
  WRITE: / 'User does not have display access.'.
ENDIF.

lv_permission =
  lv_display_user BIT-AND lif_permissions=>CO_CREATE.
IF lv_permission EQ lif_permissions=>CO_CREATE.
  WRITE: / 'User can create documents.'.
```

```
ELSE.
  WRITE: / 'User is not authorized to create documents.'.
ENDIF.
```

Listing 2.15 Mapping Permissions Using Bit Masking

As you can see, bit masking can be used as an effective compression technique. Other practical examples of bit masking include the storage of user preferences and set operations, which are described in an example in the online SAP Help Portal.

2.4 Summary

In this chapter, you learned about some advanced and perhaps lesser-known features of elementary data types in ABAP. During the course of this book, you'll see how some of these fundamental concepts provide the foundation for implementing new features in SAP NetWeaverAS ABAP, such as support for Unicode and XML processing. In the next chapter, we mix things up a bit and take a look at dynamic programming in ABAP.

Index

B

C

D

E

H

I

N

O

P

Q

R

S

Y

Z