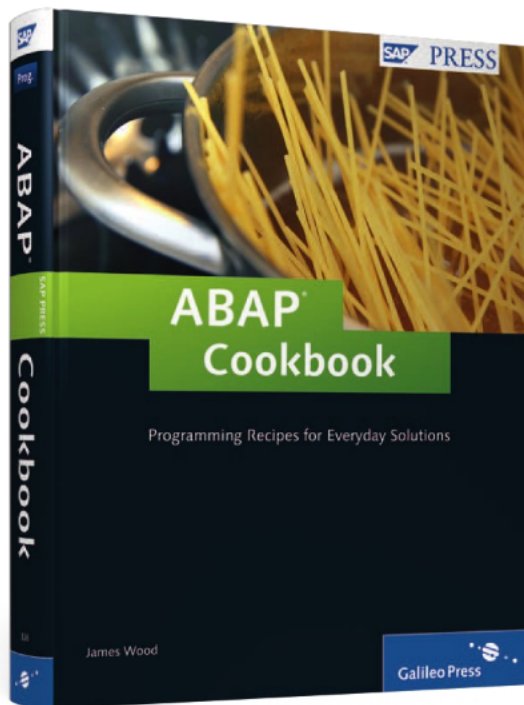


James Wood

ABAP™ Cookbook

Programming Recipes for Everyday Solutions



 Galileo Press®

Bonn • Boston

Contents at a Glance

PART I Appetizers

1	String Processing Techniques	27
2	Working with Numbers, Dates, and Bytes	57
3	Dynamic and Reflective Programming	81
4	ABAP and Unicode	109

PART II Main Courses

5	Working with Files	135
6	Database Programming	183
7	Transactional Programming	233

PART III Meals to Go

8	XML Processing in ABAP	283
9	Web Programming with the ICF	329
10	Web Services	361
11	Email Programming	393

PART IV Side Dishes

12	Security Programming	419
13	Logging and Tracing	445
14	Interacting with the Operating System	459
15	Interprocess Communication	475
16	Parallel and Distributed Processing with RFCs	511

Contents

Introduction	17
--------------------	----

PART I Appetizers

1 String Processing Techniques	27
---	-----------

1.1 ABAP Character Types	27
1.2 Designing a Custom String Library	29
1.2.1 Developing the API	29
1.2.2 Encapsulating Basic String Processing Statements	33
1.3 Improving Productivity with Regular Expressions	36
1.3.1 Understanding Regular Expressions	37
1.3.2 Regular Expression Syntax	37
1.3.3 Using Regular Expressions in ABAP	46
1.3.4 Integrating Regular Expression Support into the String Library	53
1.4 Summary	56

2 Working with Numbers, Dates, and Bytes	57
---	-----------

2.1 Numeric Operations	57
2.1.1 ABAP Math Functions	58
2.1.2 Generating Random Numbers	60
2.2 Date and Time Processing	64
2.2.1 Understanding ABAP Date and Time Types	64
2.2.2 Date and Time Calculations	65
2.2.3 Working with Timestamps	66
2.2.4 Calendar Operations	70
2.3 Bits and Bytes	73
2.3.1 Introduction to the Hexadecimal Type in ABAP	73
2.3.2 Reading and Writing Individual Bits	75
2.3.3 Bitwise Logical Operators	76
2.4 Summary	79

3	Dynamic and Reflective Programming	81
3.1	Working with Field Symbols	81
3.1.1	What Is a Field Symbol?	82
3.1.2	Field Symbol Declarations	83
3.1.3	Assigning Data Objects to Field Symbols	85
3.1.4	Casting Data Objects During the Assignment Process	89
3.2	Reference Data Objects	91
3.2.1	Declaring Data Reference Variables	91
3.2.2	Assigning References to Data Objects	93
3.2.3	Dynamic Data Object Creation	94
3.2.4	Performing Assignments Using Data Reference Variables	96
3.2.5	De-Referencing Data References	96
3.3	Introspection with ABAP Run Time Type Services	98
3.3.1	ABAP RTTS System Classes	99
3.3.2	Working with Type Objects	100
3.3.3	Defining Custom Data Types Dynamically	102
3.3.4	Case Study: RTTS Usage in the ALV Object Model	104
3.4	Dynamic Program Generation	106
3.4.1	Creating a Subroutine Pool	106
3.4.2	Creating a Report Program	107
3.4.3	Drawbacks to Dynamic Program Generation	108
3.5	Summary	108
4	ABAP and Unicode	109
4.1	Introduction to Character Codes and Unicode	109
4.1.1	Understanding Character-Encoding Systems	110
4.1.2	Limitations of Early Character-Encoding Systems	111
4.1.3	What Is Unicode?	111
4.1.4	Unicode Support in SAP Systems	113
4.2	Developing Unicode-Enabled Programs in ABAP	113
4.2.1	Overview of Unicode-Related Changes to ABAP	114
4.2.2	Thinking in Unicode	117
4.2.3	Turning on Unicode Checks	120
4.3	Working with Unicode System Classes	121

4.3.1	Converting External Data into ABAP Data Objects	121
4.3.2	Converting ABAP Data Objects into External Data Formats	124
4.3.3	Converting Between External Formats	126
4.3.4	Useful Character Utilities	129
4.4	Summary	131

PART II Main Courses

5	Working with Files	135
5.1	File Processing on the Application Server	135
5.1.1	Understanding the ABAP File Interface	136
5.1.2	Case Study: Processing Files with the ABAP File Interface ...	141
5.2	Working with Unicode	148
5.2.1	Changes to the OPEN DATASET Statement to Support Unicode	149
5.2.2	Using Class CL_ABAP_FILE_UTILITIES	149
5.3	Logical Files and Directories	150
5.3.1	Defining Logical Directory Paths and Files in Transaction FILE	151
5.3.2	Working with the Logical File API	155
5.4	File Compression with ZIP Archives	157
5.4.1	The ABAP ZIP File API	158
5.4.2	Creating a ZIP File	159
5.4.3	Reading a ZIP File	163
5.5	File Processing on the Presentation Server	167
5.5.1	Interacting with the SAP GUI via CL_GUI_FRONTEND_SERVICES	167
5.5.2	Downloading a File	168
5.5.3	Uploading a File	171
5.6	Transmitting Files Using FTP	173
5.6.1	Introducing the SAPFTP Library	173
5.6.2	Wrapping the SAPFTP Library in an ABAP Objects Class	175
5.6.3	Uploading and Downloading Files Using FTP	176
5.6.4	Implementation Details	179
5.7	Summary	182

6	Database Programming	183
6.1	Object-Relational Mapping and Persistence	183
6.1.1	Positioning of Object-Relational Mapping Tools	184
6.1.2	Persistence Service Overview	184
6.1.3	Mapping Concepts	187
6.2	Developing Persistent Classes	189
6.2.1	Creating Persistent Classes in the Class Builder	190
6.2.2	Defining Mappings Using the Mapping Assistant Tool	192
6.3	Working with Persistent Objects	198
6.3.1	Understanding the Class Agent API	199
6.3.2	Performing Typical CRUD Operations	199
6.3.3	Querying Persistent Objects with the Query Service	204
6.4	Modeling Complex Relationships	206
6.4.1	Defining Custom Attributes	207
6.4.2	Filling in the Gaps	209
6.5	Storing Text with Text Objects	214
6.5.1	Defining Text Objects	214
6.5.2	Using the Text Object API	218
6.5.3	Alternatives to Working with Text Objects	222
6.6	Connecting to External Databases	223
6.6.1	Configuring a Database Connection	223
6.6.2	Accessing the External Database	225
6.6.3	Further Reading	230
6.7	Summary	231
7	Transactional Programming	233
7.1	Introduction to the ACID Transaction Model	233
7.2	Transaction Processing with SAP LUWs	235
7.2.1	Introduction to SAP Logical Units of Work	235
7.2.2	Bundling Database Changes in Update Function Modules	239
7.2.3	Bundling Database Changes in Subroutines	242
7.2.4	Performing Local Updates	244
7.2.5	Dealing with Exceptions in the Update Task	245
7.3	Working with the Transaction Service	248
7.3.1	Transaction Service Overview	248
7.3.2	Understanding Transaction Modes	249

7.3.3	Processing Transactions in Object-Oriented Mode	253
7.3.4	Performing Consistency Checks with Check Agents	259
7.4	Implementing Locking with the Enqueue Service	262
7.4.1	Introduction to the SAP Lock Concept	262
7.4.2	Defining Lock Objects	263
7.4.3	Programming with Locks	265
7.4.4	Integration with the SAP Update System	267
7.4.5	Lock Administration	267
7.5	Tracking Changes with Change Documents	268
7.5.1	What Are Change Documents?	269
7.5.2	Creating Change Document Objects	269
7.5.3	Configuring Change-Relevant Fields	273
7.5.4	Programming with Change Documents	274
7.6	Summary	279

PART III Meals to Go

8	XML Processing in ABAP	283
8.1	Introduction to XML	283
8.1.1	What Is XML?	284
8.1.2	XML Syntax	285
8.1.3	Defining XML Documents Using XML Schema	289
8.2	Parsing XML with the iXML Library	291
8.2.1	Introducing the iXML Library API	291
8.2.2	Working with DOM	292
8.2.3	Case Study: Developing XML Mapping Programs in ABAP	297
8.2.4	Next Steps	304
8.3	Transforming XML Using XSLT	304
8.3.1	What Is XSLT?	305
8.3.2	Anatomy of an XSLT Stylesheet	305
8.3.3	Integrating XSLT with ABAP	308
8.3.4	Creating XSLT Stylesheets	308
8.3.5	Processing XSLT Programs in ABAP	310
8.3.6	Case Study: Transforming Business Partners with XSLT	311
8.3.7	Serialization of ABAP Data Objects Using asXML	314
8.4	Simple Transformation	317

8.4.1	What Is Simple Transformation?	318
8.4.2	Anatomy of a Simple Transformation Program	318
8.4.3	Learning Simple Transformation Syntax	319
8.4.4	Creating Simple Transformation Programs	324
8.4.5	Case Study: Transforming Business Partners with ST	325
8.5	Summary	327

9 Web Programming with the ICF 329

9.1	HTTP Overview	329
9.1.1	Working with the Uniform Interface	330
9.1.2	Addressability and URLs	332
9.1.3	Understanding the HTTP Message Format	333
9.2	Introduction to the ICF	335
9.3	Developing an HTTP Client Program	336
9.3.1	Defining the Service Call	337
9.3.2	Working with the ICF Client API	338
9.3.3	Putting It All Together	340
9.4	Implementing ICF Handler Modules	346
9.4.1	Working with the ICF Server-Side API	347
9.4.2	Creating an ICF Service Node	348
9.4.3	Developing an ICF Handler Class	354
9.4.4	Testing the ICF Service Node	358
9.5	Summary	360

10 Web Services 361

10.1	Web Service Overview	361
10.1.1	Introduction to SOAP	362
10.1.2	Describing SOAP-Based Services with WSDL	365
10.1.3	Web Service Discovery with UDDI	365
10.2	Providing Web Services	366
10.2.1	Creating Service Definitions	367
10.2.2	Configuring Runtime Settings	373
10.2.3	Testing Service Providers	376
10.3	Consuming Web Services	378
10.3.1	Creating a Service Consumer	379
10.3.2	Defining a Logical Port	383

10.3.3 Using a Service Consumer in an ABAP Program	386
10.4 Next Steps	391
10.5 Summary	391

11 Email Programming 393

11.1 Introduction to BCS	393
11.2 Sending Email Messages	394
11.2.1 Understanding the Simple Mail Transfer Protocol	395
11.2.2 Sending a Plain Text Message	396
11.2.3 Working with Attachments	403
11.2.4 Formatting Email Messages with HTML	408
11.3 Receiving Email Messages	411
11.3.1 Configuring Inbound Processing Rules	412
11.3.2 Processing Inbound Requests	413
11.3.3 Potential Use Cases of Inbound Processing Rules	414
11.4 Summary	416

PART IV Side Dishes

12 Security Programming 419

12.1 Developing a Security Model	419
12.1.1 Authenticating Users	420
12.1.2 Checking User Authorizations	420
12.1.3 Securing the Lines of Communication	421
12.1.4 Programming for Security	422
12.2 The SAP NetWeaver AS ABAP Authorization Concept	422
12.2.1 Overview	423
12.2.2 Developing Authorization Objects	424
12.2.3 Configuring Authorizations	430
12.2.4 Performing Authorization Checks in ABAP	433
12.2.5 Authorization Concept Review	434
12.3 Encrypting Data with ABAP	435
12.4 Performing Virus Scans	437
12.5 Protecting Web Content with CAPTCHA	438
12.5.1 What Is CAPTCHA?	439
12.5.2 Developing a CAPTCHA Component with Adobe Flex	439

12.5.3	Integrating the CAPTCHA Component with BSPs	440
12.5.4	Integrating the CAPTCHA Component with Web Dynpro	443
12.6	Summary	444

13 Logging and Tracing 445

13.1	Introducing the Business Application Log	446
13.1.1	Configuring Log Objects	446
13.1.2	Displaying Logs	448
13.1.3	Organization of the BAL API	450
13.2	Developing a Custom Logging Framework	450
13.2.1	Organization of the Class-Based API	451
13.2.2	Configuring Log Severities	452
13.3	Case Study: Tracing an Application Program	453
13.3.1	Integrating the Logging Framework into an ABAP Program	453
13.3.2	Viewing Log Instances in Transaction SLG1	456
13.4	Summary	458

14 Interacting with the Operating System 459

14.1	Programming with External Commands	459
14.1.1	Maintaining External Commands	460
14.1.2	Restricting Access to External Commands	462
14.1.3	Testing External Commands	463
14.1.4	Executing External Commands in an ABAP Program	465
14.2	Case Study: Executing a Custom Perl Script	467
14.2.1	Defining the Command to Run the Perl Interpreter	468
14.2.2	Executing Perl Scripts	469
14.3	Summary	474

15 Interprocess Communication 475

15.1	SAP NetWeaver AS ABAP Memory Organization	476
15.2	Data Clusters	477
15.2.1	Working with Data Clusters	478
15.2.2	Storage Media Types	478

15.2.3	Sharing Data Objects Using ABAP Memory	479
15.2.4	Sharing Data Objects Using the Shared Memory Buffer ...	482
15.3	Working with Shared Memory Objects	486
15.3.1	Architectural Overview	486
15.3.2	Defining Shared Memory Areas	489
15.3.3	Accessing Shared Objects	495
15.3.4	Locking Concepts	506
15.3.5	Area Instance Versioning	507
15.3.6	Monitoring Techniques	509
15.4	Summary	510
16 Parallel and Distributed Processing with RFCs		511
16.1	RFC Overview	512
16.1.1	Understanding the Different Variants of RFC	512
16.1.2	Developing RFC-Enabled Function Modules	513
16.2	Parallel Processing with aRFC	515
16.2.1	Syntax Overview	515
16.2.2	Configuring an RFC Server Group	518
16.2.3	Defining Parallel Algorithms	520
16.2.4	Case Study: Processing Messages in Parallel	522
16.3	Summary	529
The Author		531
Index		533

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.

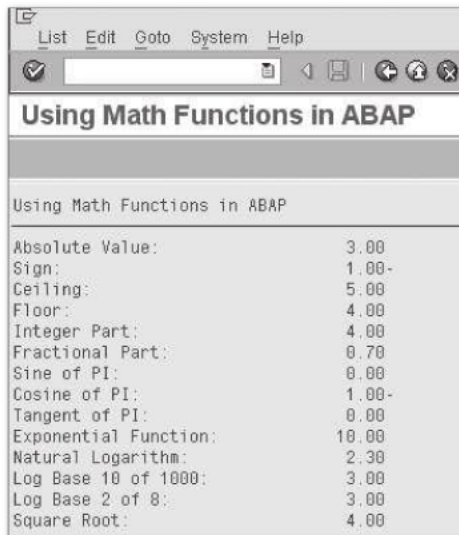
lv_result = exp( '2.3026' ).
WRITE: / 'Exponential Function:', lv_result.

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


Using Math Functions in ABAP	
Absolute Value:	3.00
Sign:	1.00-
Ceiling:	5.00
Floor:	4.00
Integer Part:	4.00
Fractional Part:	0.70
Sine of PI:	0.00
Cosine of PI:	1.00-
Tangent of PI:	0.00
Exponential Function:	10.00
Natural Logarithm:	2.30
Log Base 10 of 1000:	3.00
Log Base 2 of 8:	3.00
Square Root:	4.00

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.

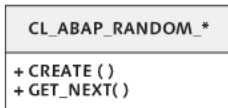


Figure 2.2 Basic UML Class Diagram for Random Number Generators

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.

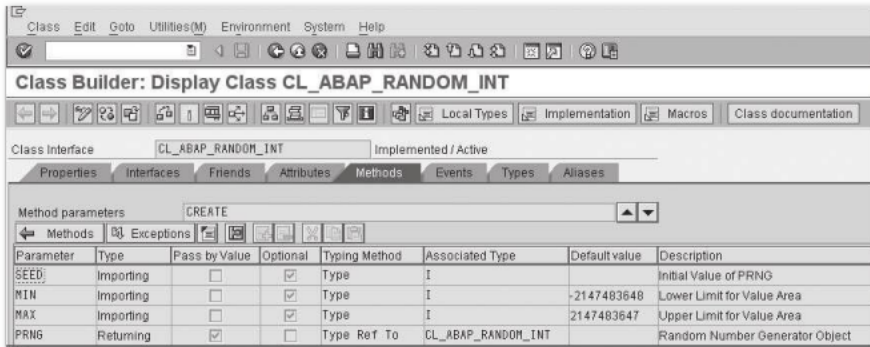


Figure 2.3 Signature of Class Method CREATE()

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 `TIMESTAMP_L` 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

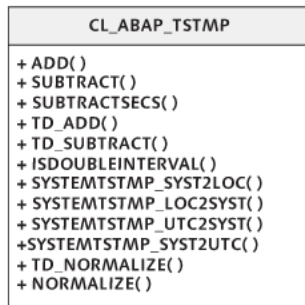


Figure 2.4 UML Class Diagram for Class `CL_ABAP_TSTMP`

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_tstamp=>add
    EXPORTING
      tstmp   = lv_tstamp
      secs    = 75
    RECEIVING
      r_tstamp = 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).

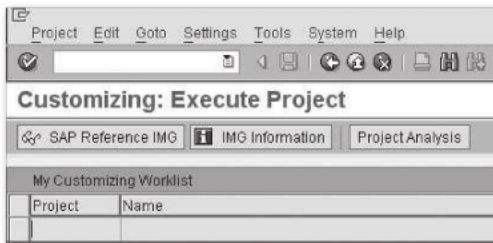


Figure 2.5 Initial Screen of Transaction SPRO

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).

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	1	0	1	0
$= (2^7 * 1) + (2^5 * 1) + (2^3 * 1) + (2^1 * 1)$ $= 170$							

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 <i>eXclusive OR</i>) 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

A

ABAP

- Basic arithmetic operators, 57*
- Built-in math functions, 58*
- Date and time processing, 64*
- Date type, 64*
- Exponentiation operator, 57*
- Hexadecimal type, 73*
- Modulus operator, 57*
- Numeric operations, 57*
- Timestamp type, 64*
- Time type, 64*
- Unicode changes, 117*
- Unicode system classes, 121*
- XSTRING type, 74*
- ABAP and Unicode, 109
- ABAP character types, 27
 - Built-in types, 27*
 - CLIKE data type, 28*
 - CSEQUENCE type, 28*
 - Static length vs. variable length types, 28*
- ABAP date and time data types, 64, 65
- ABAP Debugger, 445
- ABAP dialog programming, 237
 - Dialog step, 238*
 - Process before output event, 237*
- ABAP Dictionary
 - BLOB support, 222*
 - CLOB support, 222*
 - Enhancement categories, 119*
- ABAP Dictionary structure MATCH_RESULT, 47
- ABAP file interface, 136
 - Creating files, 141*
 - Dataset, 136*
 - Defined, 136*
 - Logical file and directory API, 155*
 - Logical files and directories, 150*
 - Reading files, 143*
 - Updating files, 145*
 - Working with Unicode, 148*
- ABAP hexadecimal type
 - BIT-AND operator, 77*
 - BIT-NOT operator, 77*
 - BIT-OR operator, 77*
 - Bitwise logical operators, 76*
 - BIT-XOR operator, 77*
 - GET BIT statement, 75*
 - Reading and writing bits, 75*
 - SET BIT statement, 75*
- ABAP math functions
 - Absolute value function, 58*
 - Base-10 logarithm function, 58*
 - Ceiling function, 58*
 - Complex expressions, 60*
 - Exponential function, 58*
 - Floor function, 58*
 - Fraction function, 58*
 - Hyperbolic trigonometric functions, 58*
 - Inverse trigonometric functions, 58*
 - Natural logarithm function, 58*
 - Sign function, 58*
 - Square root function, 58*
 - Trigonometric function, 58*
 - Truncation function, 58*
 - Usage example, 59*
- ABAP memory, 479
 - Accessibility, 480*
 - Usage example, 480*
- ABAP Objects
 - Chained method calls, 35*
 - Functional methods, 31*
 - Transient nature, 184*
- ABAP Object Services, 183
 - As an ORM tool, 184*
 - Persistence Service, 184*

- Query Service*, 198
 - Transaction Service*, 248
 - ABAP regex classes
 - Example*, 48
 - Exception types*, 51
 - UML class diagram*, 48
 - Working with submatches*, 51
 - ABAP regular expression engine, 36
 - Initial release version*, 36
 - ABAP Run Time Type Services, 98
 - ABAP Serialization XML, 314
 - asXML*, 314
 - ABAP SHIFT statement
 - Byte mode*, 75
 - ABAP string processing statements
 - IN BYTE MODE addition*, 114
 - IN CHARACTER MODE addition*, 115
 - Processing mode*, 114
 - ABAP structures
 - Alignment bytes*, 115
 - ABAP Web Service Framework
 - Advanced features*, 391
 - Creating a service consumer*, 379
 - Creating service definitions*, 367
 - Generating a service consumer call*, 387
 - Providing Web services*, 366
 - Service consumer*, 378
 - Transparency*, 389
 - Abstract class, 186
 - Accessing an external database table, 226
 - ACID transaction model, 233
 - Definition*, 233
 - Described*, 234
 - Properties*, 233
 - Adobe Flex, 439
 - Adobe Flex Framework
 - Adobe AIR runtime environment*, 359
 - Application Log Object
 - Creating*, 446
 - Area instance version, 507
 - Lifecycle*, 508
 - Area instance versioning
 - Active version*, 507
 - Area root class, 488
 - Defining*, 488
 - ASCII, 73
 - ASSIGN COMPONENT statement, 87
 - ASSIGN statement, 85
 - Basic syntax*, 85
 - CASTING addition*, 89
 - CASTING addition syntax variants*, 91
 - Asynchronous RFC
 - aRFC*, 512
 - Retrieving results*, 517
 - Synchronization with the WAIT UNTIL statement*, 516
 - Atomic commit protocol, 235
 - Authentication
 - CAPTCHA*, 438
 - Defined*, 420
 - AUTHORITY-CHECK statement, 433
 - FOR USER extension*, 434
 - Syntax*, 433
 - Authorization, 420, 423
 - Defined*, 421
 - Authorization checks, 433
 - The AUTHORITY-CHECK statement*, 433
 - Authorization fields, 426
 - Maintaining in Transaction SU20*, 426
 - Authorization objects, 423
 - Authorization fields*, 424
 - Creating a custom authorization object*, 427
 - Example*, 425
 - Maintaining in Transaction SU21*, 425
 - Overview*, 424
 - Authorization profile, 423
 - Automatic area structuring
 - Interface IF_SHM_BUILD_INSTANCE*, 502
- ## B
-
- Background RFC
 - bgRFC*, 513

BAL

- Application log object*, 446
- Application log sub-object*, 446

Basic Multilingual Plane

- BMP*, 112

Binary and hexadecimal numbers, 74

Binary number system, 74

Bit, 74

- Binary digit*, 74
- Value range*, 74

Bit masking

- Example*, 78
- Other practical examples*, 79

Bits and bytes, 73

Bitwise logical operators in ABAP, 77

BLOBS, 222

Boolean methods, 33

Boolean operators

- Truth table*, 76

Boost Regex library, 36

- John Maddock*, 36

BSPs, 357

- Class CL_HTTP_EXT_BSP*, 357

Business Address Services, 394

Business Application Log, 445

- API organization*, 450
- Configuring log severities*, 452
- Displaying logs*, 448
- Log handle*, 450
- Table BALHDR*, 446
- Transaction SLG0*, 446

Business Communication Services, 393

- BCS*, 393
- Configuration*, 394
- Inbound processing rules*, 412
- Initial release*, 393
- Receiving email messages*, 411
- Usage example*, 398
- Working with attachments*, 403

Business Server Pages

- BSPs*, 329

Business Workplace

- Transaction SBWP*, 397

Byte, 74

C

CALL FUNCTION statement

- IN UPDATE TASK addition*, 241

CALL TRANSFORMATION statement,

310

- PARAMETERS addition*, 318
- Syntax*, 310

CAPTCHA, 419, 438

- Adobe Flex component*, 439
- Defined*, 439
- Integration with BSPs*, 440
- Integration with Web Dynpro*, 443

Change document object

Creating, 269

Defined, 269

Update module, 271

Change documents, 268

- Configuring change-relevant fields*, 273
- Defined*, 269

Programming with, 269, 273, 274

Table CDHDR, 277

Table CDPOS, 277

Character codes, 109

Character-encoding system, 109

ASCII, 110

Character set, 110

Code page, 110

Defined, 109

Described, 110

EBCDIC, 111

ISO/IEC 8859, 111

Limitations of early systems, 111, 113

Check modules

- Function SXPG_DUMMY_COMMAND_CHECK*, 462

Class /BOWDK/CL_FTP_CLIENT, 175

- UML class diagram*, 175

Class /BOWDK/CL_HTML_DOCUMENT_

BCS, 409

Class /BOWDK/CL_LOGGER, 451

- UML class diagram*, 451

Class /BOWDK/CL_SAPSCRIPT_UTILS,

220

- Class /BOWDK/CL_STRING
 - Regular expression support*, 53
 - UML class diagram*, 32, 53
 - Class Builder, 33
 - Transaction SE24*, 33
 - Class CL_ABAP_CHAR_UTILITIES, 129
 - UML class diagram*, 129
 - Class CL_ABAP_CONV_IN_CE, 121
 - Stream-based processing model*, 123
 - Structure conversions*, 124
 - UML Class Diagram*, 121
 - Usage example*, 121
 - Class CL_ABAP_CONV_OUT_CE, 124
 - UML class diagram*, 124
 - Usage example*, 124
 - Class CL_ABAP_CONV_X2X_CE, 126
 - UML class diagram*, 126
 - Usage example*, 126
 - Class CL_ABAP_FILE_UTILITIES, 149
 - Class diagram*, 149
 - Description*, 150
 - Class CL_ABAP_MATCHER, 48
 - Defined*, 48
 - Class CL_ABAP_REGEX, 46
 - Defined*, 48
 - Class CL_ABAP_TSTMP
 - UML class diagram*, 69
 - Class CL_ABAP_TYPEDESCR
 - UML class diagram*, 99
 - Class CL_ABAP_VIEW_OFFLEN, 124
 - Class CL_ABAP_ZIP, 158
 - Description*, 158
 - UML class diagram*, 158
 - Class CL_BCS, 394, 396
 - And COMMIT WORK*, 398
 - Persistent class*, 396
 - Sending immediately*, 402
 - Class CL_CAM_ADDRESS_BCS, 402
 - Class CL_DISTRIBUTIONLIST_BCS, 397
 - Class CL_DOCUMENT_BCS, 398
 - Creating a text message*, 402
 - Class CL_GUI_FRONTEND_SERVICES, 167, 408
 - Method FILE_OPEN_DIALOG()*, 171
 - Method FILE_SAVE_DIALOG()*, 168
 - Method GUI_DOWNLOAD()*, 168
 - Method GUI_UPLOAD()*, 171
 - UML class diagram*, 167
 - Class CL_HTTP_CLIENT, 338
 - Class CL_IXML, 291, 292
 - Method CREATE()*, 292
 - Class CL_OS_SYSTEM, 249
 - Method INIT_AND_SET_MODES*, 250
 - Class CL_SAPUSER_BCS, 401
 - Class CX_SY_MATCHER, 51
 - Class CX_SY_REGEX, 51
 - CLOBS, 222
 - CLOSE DATASET statement, 140
 - Syntax*, 140
 - COMMIT WORK statement, 200, 220, 237
 - AND WAIT addition*, 241
 - Common Object Request Broker Architecture
 - CORBA*, 362
 - Composition technique, 61
 - Connecting to external databases, 223
 - Transaction DBCO*, 223
 - CORBA, 362
 - CREATE DATA statement, 94
 - TYPE HANDLE addition*, 94
 - CREATE DATA Statement
 - TYPE HANDLE Addition*, 100
- ## D
-
- Database programming, 183
 - CRUD operations*, 198
 - Data clusters, 477
 - Built-in statements*, 478
 - Defined*, 477

- Limitations*, 486
- Storage media types*, 478
- Data encryption, 435
- Data references, 91
 - Compared to pointers*, 92
 - Declarations*, 91
 - Declaring fully typed data references*, 92
 - De-referencing*, 92, 96
 - De-referencing generically typed data references*, 97
 - Safety precautions*, 95
- Data reference variables
 - Assignments*, 96
- Date and time calculations, 65
- Date and time operations
 - Offset/length functionality*, 66
- Date calculations
 - Example*, 66
- DELETE DATASET statement, 140
 - Permissions*, 140
 - Syntax*, 140
- DELETE statement, 478
 - Syntax*, 478
- De-referencing operator (->*), 96
- DESCRIBE FIELD statement, 87
- Document Object Model, 291
 - DOM*, 291
 - Usage example*, 292
- Document Type Definition, 289
 - DTD*, 289
- Double-byte encoding schemes
 - BIG5*, 113
 - SJIS*, 113
- Dynamic data objects, 477
- Dynamic program generation, 106
 - Creating a report program*, 107
 - Creating a subroutine pool*, 106
 - Pitfalls*, 108
- Dynamic programming, 81

E

- Email, 394
 - Formatting with HTML*, 409
- Encryption
 - Defined*, 421
- Enqueue Service, 262
- Enterprise Services Repository and Services Registry, 366
- ES Repository
 - Online Documentation*, 366
- Exception class /BOWDK/CX_FTP_EXCEPTION, 176
- Exception class CX_OS_CHECK_AGENT_FAILED, 261
- Exception class CX_OS_OBJECT_EXISTING, 200
- Exception class CX_OS_SYSTEM, 251
- EXEC SQL statement, 226
 - CONNECT Statement*, 226
 - Syntax diagram*, 226
- EXPORT statement, 478
 - Expanded syntax*, 480, 483
 - SHARED BUFFER addition*, 483
 - SHARED MEMORY addition*, 483
 - Syntax*, 478
- Extensible Markup Language
 - XML*, 283
- External commands, 459, 460
 - Check modules*, 462
 - Configuring the Perl interpreter*, 468
 - Dynamic parameters*, 462
 - Executing in ABAP*, 465
 - Executing Perl scripts*, 469
 - Function SXPB_COMMAND_EXECUTE*, 465
 - Perl*, 467
 - Python*, 467
 - Reading output*, 472
 - Restricting access*, 462
 - S_LOG_COM authorization object*, 462
 - Static parameters*, 462

Testing, 463
Transaction SM69, 460

F

Field symbols, 81

Assignments, 85, 86
Casting data objects, 89
Declaration examples, 83
Declarations, 83
Declaration scope, 83
Defined, 82
Dynamic assignments, 86
Illustration, 82
Relationship to pointers, 82
Static assignments, 85
Static assignments with offset/length specifications, 85
Typing, 83
Verifying assignments, 85
Working with internal tables, 88
Working with structures, 87

File processing on the application server, 135

File processing on the presentation server, 167

Downloading a file, 168
Uploading a file, 171

File Transfer Protocol, 135, 173

FTP, 173
Secure FTP, 175

FIND statement

Example, 46
Syntax, 46

Function BAL_DB_SAVE, 450

Function BAL_LOG_CREATE, 450

Function BAL_LOG_EXCEPTION_ADD, 450

Function BAL_LOG_MSG_ADD, 450

Function BAL_LOG_MSG_ADD_FREE_TEXT, 450

Function CHANGEDOCUMENT_READ, 278

Function DB_COMMIT, 237

Function DELETE_TEXT, 222

Function FILE_GET_NAME, 155

Usage Example, 155

Function FILE_GET_NAME_AND_LOGICAL_PATH, 155

Function FILE_GET_NAME_USING_PATH, 155

Function FTP_CLIENT_TO_R3, 174

Function FTP_COMMAND, 174

Function FTP_CONNECT, 174

Usage Example, 179

Function FTP_DISCONNECT, 174

Usage example, 181

Function FTP_R3_TO_CLIENT, 174

Function FTP_R3_TO_SERVER, 174

Usage example, 180

Function FTP_SERVER_TO_R3, 174

Function group GRAP, 167

Function group SFIL, 155

Function group SFTP, 174

Function GUID_CREATE, 201

Function MASTER_IDOC_DISTRIBUTE, 521

Function READ_TEXT, 221

Function SAVE_TEXT, 218

Function SCMS_BINARY_TO_XSTRING, 408

Function SCMS_XSTRING_TO_BINARY, 159, 163

Function SPBT_INITIALIZE, 521

Function SXPG_COMMAND_EXECUTE, 465

G

GENERATE SUBROUTINE POOL statement, 106

GET DATASET statement, 146

Syntax, 146

GET REFERENCE OF statement, 93
Example, 93
 GUID, 187
Globally Unique Identifier, 187

H

Hexadecimal number system, 74
 HTML, 284
Example, 284
 HTML entity references, 44
 HTTP, 329
Addressability and URLs, 332
Common request methods, 331
DELETE method, 331
Example client program, 336
GET method, 331
Header fields, 333
HEAD method, 331
Hypertext Transfer Protocol, 329
Message format, 333
Overview, 329
POST method, 331
PUT method, 331
Relationship to the TCP/IP , 333
Request entity body, 334
Response entity body, 334
Transport protocol, 333
Uniform interface, 330

I

ICF, 329
Accessing URL query string parameters, 355
Activating services, 354
Client API, 338
Configuring basic authentication, 351
Debugging with the ABAP Debugger, 358

Defining service nodes in Transaction SICF, 348
Developing an ICF handler class, 354
Handler modules, 346
Interface IF_HTTP_CLIENT, 338
Interface IF_HTTP_EXTENSION, 348
Interface IF_HTTP_SERVER, 348
Internet Communication Framework, 329
Introduction, 335
Positioning, 336
Service nodes, 348
Testing ICF service nodes, 358
Virtual hosts, 348
 ICF handler module
Flow return code, 358
 ICM
Functionality, 335
Internet Communication Manager, 335
Positioning, 335
 IDocs, 363
 Implicit database commits, 237
 IMPORT statement, 478
Syntax, 478
 Information Age, 27
 INSERT REPORT statement, 107
 Integration testing, 445
 Interface description language
IDL, 363
 Interface IF_DOCUMENT_BCS, 398
 Interface IF_HTTP_CLIENT, 338
 Interface IF_HTTP_EXTENSION
Method HANDLE_REQUEST(), 348
 Interface IF_HTTP_REQUEST, 338
 Interface IF_HTTP_RESPONSE, 339
 Interface IF_INBOUND_EXIT_BCS, 412
Implementation example, 414
 Interface IF_IXML, 292
 Interface IF_IXML_DOCUMENT, 311
Method CREATE_SIMPLE_ELEMENT(), 297
 Interface IF_IXML_ISTREAM, 302, 310
 Interface IF_IXML_NODE, 310

Interface IF_IXML_OSTREAM, 311
 Interface IF_IXML_PARSER, 302
 Interface IF_IXML_STREAM_FACTORY,
 302
 Interface IF_MAPPING, 298
 EXECUTE() method, 299
 Interface IF_OS_CHECK, 259
 Interface IF_OS_FACTORY, 203
 Interface IF_OS_TRANSACTION, 249
 Methods, 249
 Interface IF_OS_TRANSACTION_
 MANAGER, 249
 Interface IF_RECIPIENT_BCS, 397
 Interface IF_SENDER_BCS, 394, 397
 Interface IF_SERIALIZABLE_OBJECT,
 315, 489
 Usage example, 315
 Interface IF_SHM_BUILD_INSTANCE,
 489, 502
 Intermediate Documents, 363
 IDocs, 363
 Internal tables
 Header lines, 88
 Using assigned work areas, 89
 Internet Message Access Protocol
 IMAP, 395
 Interprocess communication, 475
 Introspection, 81
 iXML library, 291
 Implementation, 291
 Release, 291
 iXML library API, 291
 UML class diagram, 292

J

Java, 298

K

Kernel methods, 291

L

LOAD-OF-PROGRAM event, 251
 Local Data Queue
 LDQ, 513
 Locators and Streams API, 223
 Lock object
 As a logical lock, 263
 Dequeue function, 265
 Enqueue function, 265
 Lock Mode, 264
 Lock modules, 265
 Ownership, 267
 Lock objects, 263
 Defining, 263
 Foreign lock exceptions, 266
 Logging, 445
 Logical port, 383
 Configuration type, 385
 Defining in Transaction LPCONFIG,
 384
 Defining in Transaction
 SOAMANAGER, 384
 Editing in Transaction SOAMANAGER,
 386
 Setting the default port, 385
 Logical unit of work
 Lifecycle, 235
 LUW, 235
 LOOP AT statement
 ASSIGNING addition, 89
 Lvalue, 97

M

Mapping Assistant
 Business key assignment type, 194
 Class identifier assignment type, 194
 Creating a persistence map, 192
 GUID assignment type, 194
 Object reference assignment type, 194
 Value attribute assignment type, 194

Markup language, 284
 Defined, 284
 HTML, 284
 MathML, 284
 Message digest
 ABAP implementation, 436
 Defined, 435
 Message digests
 Encrypting passwords, 436
 Function MD5_CALCULATE_HASH_
 FOR_CHAR, 436
 Function MD5_CALCULATE_HASH_
 FOR_RAW, 437

N

Native SQL, 223
 ABAP Keyword Documentation, 230
 Numeric wrapper class, 76

O

Object-oriented programming
 Factory pattern, 61
 Object-oriented transactions
 Creating, 251
 Object-relational mapping, 183
 Benefits, 184
 Mapping, 184
 ORM, 184
 OLTP systems, 64
 OPEN DATASET statement, 136
 Access mode, 136
 ENCODING DEFAULT addition, 143,
 149
 Error handling, 138
 File permissions, 138
 NON-UNICODE addition, 149
 Storage mode, 137
 Syntax, 136

 Unicode changes, 149
 UTF-8 addition, 149
 WITH SMART LINEFEED addition,
 143
 Open SQL, 183
 DELETE statement, 199
 INSERT statement, 199
 SELECT statement, 199
 UPDATE statement, 199
 Operating system, 459

P

Package SIXXML_TEST, 304
 Paging buffer, 477
 Parallel processing, 511
 Case study, 522
 Class /BOWDK/CL_PBT_UTILITIES,
 523
 Designing algorithms, 520
 Initializing the PBT environment, 523
 With RFCs, 515
 With the aRFC interface, 520
 PERFORM statement
 ON COMMIT addition, 242
 ON ROLLBACK addition, 244
 Perl, 467
 Persistence, 183
 Persistence classes
 Agent classes, 185
 Persistence map
 Assignment types, 194
 Persistence mapping
 By business key, 187
 By instance-GUID, 187
 By instance-GUID and business key,
 188
 Multiple-table mapping, 188
 Single-table mapping, 188
 Strategies, 187
 Structure mappings, 188

Persistence Service, 184
 Class agent API, 199
 Layer of abstraction, 185
 Managing persistent objects, 185
 Mapping concepts, 187
 Mapping strategies, 187
 Multiple-table mapping, 188
 Overview, 184
 Persistent class, 185
 Persistent objects, 184
 Single-table mapping, 188
 Structure mappings, 188
 Support for other storage media, 188

Persistent classes, 185
 Creating, 187, 189, 198, 206
 Creating in the Class Builder, 190
 Instantiation context, 187
 Mapping Assistant tool, 192
 Mapping by business key, 187
 Mapping to a persistence model, 184
 Mapping by instance-GUID, 187
 Mapping types, 187
 UML class diagram, 185

Persistent objects
 Creating, 200
 Deleting, 203
 Managed objects, 186
 Reading, 201
 Updating, 202
 Working with, 187, 198

Pointers
 Defined, 82
 De-referencing pointers, 82
 Relationship to a data object, 92

Post Office Protocol
 POP, 395

Process before output
 PBO, 237

Programming with external commands,
 459

Q

Query Service, 198, 204
 Queued RFC
 qRFC, 513

R

Random number generators, 61
 Class CL_ABAP_RANDOM, 61
 Class CL_ABAP_RANDOM_INT, 61
 Seed, 61
 Usage example, 62

Random numbers, 60
 Generating, 60

READ DATASET statement, 139
 ACTUAL LENGTH addition, 140
 MAXIMUM LENGTH addition, 140
 Syntax, 139

READ TABLE statement
 ASSIGNING addition, 89

RECEIVE statement, 517

Reference data objects, 91

Reflective programming, 81

Regular expressions, 27, 36
 ABAP regular expression classes, 46
 Backreferences, 42
 Basic metacharacters, 37
 Boost Regex library, 36
 Character class, 41
 FIND statement, 46
 Formatting URLs, 44
 Ignoring case, 51
 Lookahead, 45
 Matching ABAP variable names, 40
 Matching a word boundary, 41
 Metacharacter, 37
 Negative lookahead, 45
 Parsing delimited file records, 43
 Positioning, 37
 Positive lookahead, 45

- POSIX-style regular expressions*, 36
 - Regexes*, 40
 - REPLACE statement*, 46
 - Searching for HTML markup*, 41
 - Syntax*, 37
 - Testing with DEMO_REGEX_TOY*, 52
 - Using ABAP regex classes*, 48
 - Using quantifiers*, 41
 - Using regexes in the FIND and REPLACE statements*, 46
 - Using regular expressions in ABAP*, 46
 - Remote function call
 - RFC*, 362
 - Remote method invocation
 - RMI*, 362
 - Remote procedure call
 - RPC*, 362
 - REPLACE statement
 - Example*, 48
 - Syntax*, 47
 - REST
 - Representational State Transfer*, 336
 - RESTful Web Services, 336, 361
 - RFC interface, 511
 - RFCs, 511
 - Asynchronous call*, 515
 - Example*, 513
 - Finding*, 514
 - Overview*, 512
 - Variants*, 512
 - RFC server group, 518
 - Example*, 519
 - Maintaining in Transaction RZ12*, 519
 - Roles, 423
 - ROLLBACK WORK statement, 238
 - RTTS, 99
 - Class CL_ABAP_TABLEDESCR*, 100
 - Class CL_ABAP_TYPEDESCR*, 99
 - Class hierarchy*, 99
 - Common uses*, 106
 - Creating a custom elementary type*, 102
 - Creating a Custom Structure Type*, 102
 - Creating data objects dynamically*, 100
 - System classes*, 99
 - Usage in the ALV object model*, 104
 - Rvalue, 97
- ## S
-
- SAP Business Suite, 64
 - SAP Calendar, 70
 - API functions*, 72
 - Configuration*, 72
 - Maintenance*, 71
 - SAP Customizing implementation guide, 71
 - Transaction SPRO*, 71
 - SAPFTP library, 173
 - Report program RSFTP002*, 174
 - Report program RSFTP005*, 174
 - SAP Interactive Forms, 415
 - SAP List Viewer, 104
 - ALV*, 104
 - ALV Object Model*, 104
 - Dynamic creation of field catalog*, 104
 - Field catalog*, 104
 - SAP Lock Concept, 262
 - Integration with the SAP update system*, 267
 - Introduction*, 262
 - Lock administration*, 267
 - SAP LUW, 235, 250
 - Bundling changes in subroutines*, 242
 - Defined*, 238
 - Introduction*, 235
 - Local updates*, 244
 - Update function modules*, 239
 - SAP MaxDB, 225
 - SAP NetWeaver AS ABAP, 236
 - As a preemptive multitasking system*, 236
 - Basic architecture*, 236
 - Context switching*, 238
 - Update work process*, 238

- SAP NetWeaver AS ABAP authorization concept, 419, 422
 - Authorization*, 423
 - Authorization object*, 423
 - Authorization profile*, 423
 - Authorizations*, 430
 - Overview*, 423
 - Roles*, 423
 - Summary*, 434
- SAP NetWeaver AS ABAP memory organization, 476
 - Illustration*, 476
 - Local memory*, 476
 - Shared memory*, 476
- SAP NetWeaver Process Integration, 297
 - Description*, 297
 - SAP PI*, 297
- SAPscript text object
 - Text header*, 218
- SAPscript text object instances
 - Creating*, 218
 - Deleting*, 222
 - Reading*, 221
 - Updating*, 221
- SAPscript text objects, 214
 - Alternatives*, 222
 - API*, 218
 - Defining*, 214, 218
 - Text IDs*, 214
- Secure Network Communications
 - SNC*, 421
- Security model, 419
 - Key elements*, 420
- Security programming, 419
 - Authentication*, 420
 - Authorization*, 420
 - Design points*, 422
 - Developing a security model*, 419
 - Encryption*, 421
 - Least privilege principle*, 422
 - Performing authorization checks*, 433
 - Virus scans*, 437
- Security roles, 430
 - Maintaining in Transaction PFCG*, 430
- Service consumer
 - ABAP proxy class*, 383, 388
 - Binding to a WSDL file*, 381
 - Design-time repository object*, 383
 - Editing in the Object Navigator*, 383
 - Example*, 389
 - Logical port*, 383
 - Selecting a prefix*, 381
 - Usage scenario in ABAP*, 386
 - Viewing an ABAP proxy class*, 389
- Service definition, 367
 - Assigning to a transport request*, 370
 - Configuring runtime settings*, 373
 - Creating with the Service Wizard*, 367
 - Deploying*, 370
 - Editing an endpoint*, 375
 - Editing in the Object Navigator*, 372
 - Name mapping*, 370
- Service-oriented architecture, 361
 - SOA*, 361
- Service provider
 - Authentication*, 375
 - Downloading a WSDL file*, 373
 - Testing*, 376
 - Transport guarantee*, 375
- Service Wizard
 - Accessing in the Object Navigator*, 367
- SET DATASET statement
 - Syntax*, 146
- SET UPDATE TASK LOCAL statement, 244
- Shared memory, 475
 - Extended memory buffer*, 477
 - Paging buffer*, 477
 - Roll buffer*, 477
 - SAP buffer*, 477
- Shared memory area, 486
 - Area handle*, 487
 - Area instance versioning*, 507
 - Automatic area structuring*, 502
 - Basic properties*, 490
 - Defined*, 487

- Defining in Transaction SHMA*, 486
- Dynamic properties*, 493
- Fixed properties*, 493
- Monitoring in Transaction SHMM*, 509
- Naming conventions*, 489
- Runtime settings*, 494
- Shared memory area instance
 - Versioning*, 487
- Shared memory areas
 - Defining*, 489
- Shared memory objects, 486
 - Abstracting the API*, 505
 - API usage*, 495
 - Architecture*, 486
 - Area class*, 486
 - Area root class*, 486
 - Locking concepts*, 506
 - Read lock*, 506
 - Shared memory area*, 486
 - UML class diagram of base components*, 486
 - Update lock*, 506
 - Write lock*, 506
- Simple API for XML, 291
 - SAX*, 291
- Simple Mail Transfer Protocol, 395
 - Defined*, 395
 - SMTP*, 395
- Simple object access protocol, 362
 - SOAP*, 362
- Simple Transformation, 317, 409
 - ABAP data binding*, 319
 - Addressing data roots*, 321
 - Basic syntax*, 325
 - Creating ST programs*, 324
 - Data roots*, 320
 - Defined*, 318
 - Deserialization*, 318
 - Flow control commands*, 322
 - Main template*, 318
 - Serialization*, 318
 - ST*, 318
 - Symmetry*, 323
- <tt*
 - attribute> command*, 327
 - cond> command*, 322
 - cond-var> command*, 322
 - deserialize> command*, 323
 - group> command*, 323
 - loop> command*, 323, 327
 - serialize> command*, 323
 - skip> command*, 322
 - switch> command*, 322
 - switch-var> command*, 322
 - value> command*, 320
 - Usage example*, 325
- SOA, 361, 365
 - Web Services*, 361
- SOAP, 362
 - Comparison to legacy protocols*, 362
 - Defined*, 362
 - HTTP*, 363
 - Introduction*, 362
 - Language independence*, 362
 - Message flow*, 364
 - Message structure*, 363
 - Platform independence*, 362
 - Service Description Language*, 365
 - Transport layer protocol*, 363
 - Using SMTP*, 415
 - XML message format*, 362
- soapUI, 376
 - Building a SOAP request*, 377
 - Configuring basic authentication*, 377
 - Running a test*, 378
- SPLIT statement, 43
- SQL, 183
- String processing techniques, 27
 - Built-in statements*, 29
- String testing, 445
- Structure component de-referencing operator, 97
- Structure component selector operator, 87
- Structure THEAD, 218
- Structure TLINE, 218

Synchronous RFC
sRFC, 512

T

Table VBLOG, 238
 Tag interface, 315
 Text files vs. binary files, 137
 Time calculations
 Example, 66
 Timestamps, 66
 Class *CL_ABAP_TSTMP*, 66
 Conversion, 67
 CONVERT statement, 67
 Daylight savings time, 67
 GET TIME STAMP statement, 67
 Operations using *CL_ABAP_TSTMP*, 69
 Retrieving system time, 67
 TIMESTAMPL type, 66
 TIMESTAMP type, 66
 UTC format, 64
 Tracing, 445
 Transactional programming, 233
 Transactional RFC
 tRFC, 513
 Transaction /BOWDK/LOG_CONF, 452
 Transaction DBCO, 223
 Creating a database connection, 224
 Transaction FILE, 151
 Creating a logical file path, 152
 Physical path assignment, 152
 Transaction SCOT, 412
 Transaction SE75, 214
 Transaction SE93, 251
 Transaction Service, 248
 Check agents, 259
 Compatibility mode, 250
 Listening for transaction events, 258
 Object-oriented mode, 250
 Subtransactions, 257
 Transaction manager, 249
 Transaction mode, 249

Typical usage scenario, 257
 UML class diagram, 249
 Update mode, 250
 Transaction SHMA, 486
 Transaction SICF, 348
 Transaction SLG0, 446
 Transaction SLG1, 448
 Transaction SM12, 267
 Transaction SM13, 245
 Transaction SM69, 460
 Transaction SOAMANAGER, 373
 Access the WSDL document for a service, 373
 Service Configuration Editor, 373
 TRANSFER statement, 138
 Class-based exceptions, 139
 LENGTH addition, 139
 NO END OF LINE addition, 139
 Syntax, 138
 Two's complement notation, 76

U

UDDI, 365, 366
 Description and discovery process, 366
 Service registry, 366
 UML, 32
 Class diagram, 32
 Unicode, 73, 109, 148
 ABAP development, 113
 Basic Multilingual Plane, 112
 Code point, 110
 Code point conversions, 130
 Defined, 111
 Impacts to structure operations in ABAP, 115
 Support in SAP systems, 113
 Thinking in Unicode, 117
 Turning on Unicode checks, 120
 Unicode-related changes to ABAP, 114
 Using structured fields as character types, 117

- Unit testing, 445
- Universal Description, Discovery, and Integration, 366
 - UDDI, 366
- Update function module
 - Creating, 239
 - Processing options, 239
- Update function modules
 - Restrictions, 240
- Update request log, 245
 - Deleting entries, 246
 - Transaction SM13, 245
- Update Request Log
 - Repeating an update, 246
- Update task, 238
 - Dealing with exceptions, 240, 242, 245
- URLs
 - Basic syntax, 332
 - Encoding with class `CL_HTTP_UTILITY`, 345
 - Host name, 332
 - Path, 333
 - Port, 332
 - Protocol specifier, 332
 - Query string, 333
 - URL encoding, 345
- URLs, 332
- UTF-8, 112
- UTF-16, 112
 - Default usage in SAP systems, 114
 - Surrogate pairs, 112
- UTF-32, 112

V

- Variability analysis, 81
- Variable-length encoding scheme
 - UTF-8, 112
 - UTF-16, 112
 - UTF-32, 112
- Variable-length encoding schemes, 112

- Virus Scan Interface, 437
 - Class `CL_VSI`, 437
 - Usage example, 437

W

- W3C, 305
- WAIT UNTIL statement, 517
- WDA, 357
 - Class `CL_WDR_MAIN_TASK`, 357
- Web Dynpro for ABAP
 - WDA, 329
- Web programming, 329
 - Human web, 329
 - Programmable web, 329
- Web Service Navigator, 376
- Web services, 361
 - ABAP Web Service Framework, 361
 - Consuming in ABAP, 378
 - Defined, 361
 - Discovery with UDDI, 365
 - Next steps, 391
 - Overview, 361
 - Providing in ABAP, 366
 - Proxy objects, 365
 - Recommended reading, 391
 - Self-describing, 365
 - Service registry, 366
 - SOAP, 362
- Web Services Description Language, 365
 - WSDL, 365
- World Wide Web, 27, 329
- WSDL, 365
 - Client usage, 365
 - Generation, 365
 - Type declarations, 365

X

- XHTML, 284
 - Extensible Hypertext Markup Language, 409

XML, 283

- Comments*, 288
- Data modeling*, 285
- Defined*, 283, 284
- Defining attributes*, 287
- Defining elements*, 286
- Element naming rules*, 286
- Empty element*, 286
- Entity references*, 288
- Extensible Markup Language*, 283
- Format*, 285
- Introduction*, 283
- Meta-markup language*, 284
- Namespace*, 306
- Openness*, 285
- Parsing*, 291
- Processing instructions*, 287
- Processing models*, 291
- Root element*, 286
- Schema definition*, 289
- Self-describing documents*, 285
- Syntax*, 285
- Syntax example*, 285
- Unicode encoding*, 285
- Usage in Web services*, 285

XML documents

- Validity*, 289

XML processing in ABAP, 283

XML Schema, 289, 365

- Constraints*, 289
- Example*, 290
- Use in standards*, 289

XPath, 306

- Location path*, 306
- Location steps*, 306
- Specification*, 306

XSLT, 304

- Anatomy of a stylesheet*, 307
- Calling ABAP modules in a stylesheet*, 311
- Creating XSLT programs*, 308
- Declarative approach*, 305
- Exceptions*, 311
- Extensible Stylesheet Language Transformations*, 304
- Literal result elements*, 307
- Matching template rules*, 307
- Processor*, 305
- Resources*, 304
- SAP XSLT Processor Reference*, 308
- Specification*, 306
- Stylesheet*, 305
- Support release*, 308
- Template rules*, 305
- Testing XSLT programs*, 313
- Transformation*, 305
- Transformation Editor*, 309, 313
- Transformation process*, 305

Y

Yahoo! Geocoding Web Service, 336

Z

ZIP archive files, 158

- Creation example*, 159
- Reading example*, 163