Contents at a Glance

1 Introduction ............................................................... 17
2 General Basic Rules ................................................... 23
3 ABAP-Specific Basic Rules ........................................... 41
4 Structure and Style ..................................................... 79
5 Architecture ............................................................... 155
6 Secure and Robust ABAP .............................................. 217
A Obsolete Language Constructs ..................................... 341
B Automatic Check of Naming Conventions .......................... 365
C Table of Rules ........................................................... 373
D Recommended Reading ................................................ 377
E The Authors ............................................................... 379
Contents

Foreword ........................................................................................................... 13
Acknowledgments .......................................................................................... 15

1 Introduction ................................................................................................. 17
  1.1 What Are Programming Guidelines? ..................................................... 17
  1.2 Why Programming Guidelines? ............................................................. 18
  1.3 Which Guidelines Are Involved Here? .................................................. 18
  1.4 Target Audience ................................................................................ 19
  1.5 How to Use This Book ...................................................................... 20

2 General Basic Rules .................................................................................. 23
  2.1 Separation of Concerns ...................................................................... 23
  2.2 KISS Principle ................................................................................... 32
  2.3 Correctness and Quality ..................................................................... 34

3 ABAP-Specific Basic Rules ......................................................................... 41
  3.1 ABAP Objects as a Programming Model ............................................ 41
  3.2 Program Type and Program Attributes ............................................. 50
    3.2.1 Program Type ........................................................................ 51
    3.2.2 Program Attributes ................................................................ 55
    3.2.3 Original Language .................................................................. 60
  3.3 Modern ABAP ..................................................................................... 62
  3.4 Checks for Correctness ....................................................................... 65
    3.4.1 Syntax Check ........................................................................ 65
    3.4.2 Extended Program Check ....................................................... 69
    3.4.3 Code Inspector ....................................................................... 72
    3.4.4 ABAP Test Cockpit ............................................................... 76
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.2</td>
<td>Modularization</td>
<td>157</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Static Classes and Singletons</td>
<td>161</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Inheritance</td>
<td>166</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Class References and Interface References</td>
<td>167</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Local Types for Global Classes</td>
<td>169</td>
</tr>
<tr>
<td>5.1.7</td>
<td>Instance Constructor</td>
<td>171</td>
</tr>
<tr>
<td>5.2</td>
<td>Error Handling</td>
<td>172</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Reaction to Error Situations</td>
<td>172</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Classical and Class-Based Exceptions</td>
<td>174</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Exception Categories</td>
<td>178</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Exception Texts</td>
<td>180</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Using Exception Classes</td>
<td>183</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Handling and Propagating Exceptions</td>
<td>185</td>
</tr>
<tr>
<td>5.2.7</td>
<td>Cleanup After Exceptions</td>
<td>186</td>
</tr>
<tr>
<td>5.2.8</td>
<td>Catchable Runtime Errors</td>
<td>188</td>
</tr>
<tr>
<td>5.2.9</td>
<td>Assertions</td>
<td>190</td>
</tr>
<tr>
<td>5.2.10</td>
<td>Messages</td>
<td>191</td>
</tr>
<tr>
<td>5.3</td>
<td>User Interfaces</td>
<td>195</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Selecting the User Interface Technology</td>
<td>195</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Encapsulating Classical User Interfaces</td>
<td>199</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Lists</td>
<td>204</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Accessibility</td>
<td>207</td>
</tr>
<tr>
<td>5.4</td>
<td>Data Storage</td>
<td>208</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Persistent Data Storage</td>
<td>208</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Database Accesses</td>
<td>210</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Client Handling</td>
<td>211</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Using the Shared Memory</td>
<td>213</td>
</tr>
</tbody>
</table>

### 6 Secure and Robust ABAP

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Data Types and Data Objects</td>
<td>217</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Bound and Standalone Data Types</td>
<td>218</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Declaration of Data Types and Constants</td>
<td>220</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Declaration of Variables</td>
<td>224</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Including Structures</td>
<td>226</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Using Types</td>
<td>228</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Referring to Data Types or Data Objects</td>
<td>230</td>
</tr>
<tr>
<td>6.1.7</td>
<td>Table Work Areas</td>
<td>232</td>
</tr>
<tr>
<td>6.1.8</td>
<td>Literals</td>
<td>233</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>6.1.9</td>
<td>Strings</td>
<td>236</td>
</tr>
<tr>
<td>6.1.10</td>
<td>Start Values</td>
<td>238</td>
</tr>
<tr>
<td>6.1.11</td>
<td>Data Objects for Truth Values</td>
<td>239</td>
</tr>
<tr>
<td>6.2</td>
<td>Assignments, Calculations, and Other Accesses to Data</td>
<td>241</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Assignments Between Different Types</td>
<td>241</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Avoiding Invalid Values</td>
<td>243</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Using Conversion Rules</td>
<td>245</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Specification of Numbers</td>
<td>247</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Selecting the Numeric Type</td>
<td>249</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Rounding Errors</td>
<td>253</td>
</tr>
<tr>
<td>6.2.7</td>
<td>Division by Zero</td>
<td>255</td>
</tr>
<tr>
<td>6.2.8</td>
<td>Casting</td>
<td>255</td>
</tr>
<tr>
<td>6.2.9</td>
<td>Runtime Errors When Accessing Data Objects</td>
<td>257</td>
</tr>
<tr>
<td>6.2.10</td>
<td>Anonymous Containers</td>
<td>259</td>
</tr>
<tr>
<td>6.2.11</td>
<td>Passing Global Data by Reference</td>
<td>260</td>
</tr>
<tr>
<td>6.3</td>
<td>System Fields</td>
<td>262</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Access</td>
<td>262</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Obsolete and Internal System Fields</td>
<td>264</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Evaluation</td>
<td>265</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Return Value</td>
<td>267</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Using System Fields as Actual Parameters</td>
<td>268</td>
</tr>
<tr>
<td>6.3.6</td>
<td>Using System Fields on the User Interface</td>
<td>270</td>
</tr>
<tr>
<td>6.3.7</td>
<td>Using System Fields in Operand Positions</td>
<td>272</td>
</tr>
<tr>
<td>6.4</td>
<td>Internal Tables</td>
<td>273</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Selecting the Table Category</td>
<td>274</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Secondary Keys</td>
<td>276</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Initial Memory Allocation</td>
<td>280</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Sorted Filling</td>
<td>281</td>
</tr>
<tr>
<td>6.4.5</td>
<td>Aggregated Filling</td>
<td>283</td>
</tr>
<tr>
<td>6.4.6</td>
<td>Output Behavior</td>
<td>284</td>
</tr>
<tr>
<td>6.4.7</td>
<td>Loop Processing</td>
<td>286</td>
</tr>
<tr>
<td>6.5</td>
<td>Modularization Units</td>
<td>287</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Function Modules and Subroutines</td>
<td>288</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Type of the Formal Parameters of Procedures</td>
<td>289</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Transfer Type of Formal Parameters</td>
<td>292</td>
</tr>
<tr>
<td>6.5.4</td>
<td>Passing Output Parameters by Reference</td>
<td>294</td>
</tr>
<tr>
<td>6.5.5</td>
<td>Typing of Formal Parameters</td>
<td>296</td>
</tr>
<tr>
<td>6.5.6</td>
<td>Internal and External Procedure Calls</td>
<td>298</td>
</tr>
<tr>
<td>6.5.7</td>
<td>Exiting Procedures</td>
<td>302</td>
</tr>
</tbody>
</table>
Contents

6.5.8 Dialog Modules and Event Blocks .................................. 304
6.5.9 Macros ........................................................................... 306

6.6 Dynamic Programming Techniques .............................................. 309
   6.6.1 Using Dynamic Programming Techniques ....................... 309
   6.6.2 Runtime Errors During Dynamic Processing .................... 311
   6.6.3 Using Dynamic Data Objects .......................................... 313
   6.6.4 Memory Consumption of Dynamic Memory Objects ....... 315
   6.6.5 Administration Costs of Dynamic Memory Objects ...... 318
   6.6.6 Accessing Data Objects Dynamically .............................. 321
   6.6.7 Generic Programming ..................................................... 324

6.7 Internationalization ..................................................................... 329
   6.7.1 Storing System Texts ....................................................... 329
   6.7.2 Translation-Friendly Message Texts ................................. 331
   6.7.3 Text Environment ........................................................... 333
   6.7.4 Character Set of Source Code ......................................... 335
   6.7.5 Splitting Texts ................................................................. 336
   6.7.6 Codepages for Files ........................................................ 337

Appendices ..................................................................................... 339

A Obsolete Language Constructs ............................................................. 341
   A.1 Procedures .................................................................................. 342
   A.2 Declarations ................................................................................ 343
   A.3 Object Generation ...................................................................... 347
   A.4 Calls and Exits ............................................................................. 347
   A.5 Program Flow Control ................................................................. 349
   A.6 Assignments ............................................................................... 350
   A.7 Calculation Statements ............................................................ 352
   A.8 Processing Character and Byte Strings ........................................ 353
   A.9 Internal Tables ............................................................................ 355
   A.10 Dynpro Flow Logic ................................................................. 357
   A.11 Classical List Processing .......................................................... 358
   A.12 Data Storage ........................................................................... 360
   A.13 Contexts .................................................................................. 362
   A.14 External Interfaces ..................................................................... 363

B Automatic Check of Naming Conventions ............................................ 365
   B.1 Naming Conventions in the Code Inspector .......................... 365
   B.2 Type-Specific Prefix Components ............................................ 366
   B.3 Prefixes for Procedure-Local Declarations ................................ 367
## Contents

- B.4 Structured Programming ............................................................. 369
- B.5 Object-Oriented Programming .................................................... 370
- B.6 Assessment of the Naming Conventions ...................................... 371
- C Table of Rules ....................................................................................... 373
- D Recommended Reading ....................................................................... 377
- E The Authors ......................................................................................... 379

Index ........................................................................................................... 381
“If you wanted to know how much I loved you, you would have to invent a completely new language.”
—Friedrich von Schiller

3 ABAP-Specific Basic Rules

Besides the rules specified in Chapter 2, General Basic Rules, this chapter additionally introduces you to a set of ABAP-specific basic rules that result from the special technical conditions of the ABAP language, the ABAP runtime environment, and its history. These basic rules also determine many of the more specific rules that follow this chapter.

3.1 ABAP Objects as a Programming Model

Background

ABAP is a hybrid programming language that supports both a procedural and an object-oriented programming model. The procedural programming model is based on the modularization of programs in classical processing blocks, that is, event blocks, dialog modules, function modules, and subroutines. In ABAP Objects, the class conceptually supersedes the classical program,1 and the modularization is implemented through its methods.

Both models are interoperable in such a way that you can access classes in classical processing blocks and call classical programs and procedures within methods. The hybrid nature of the language is mainly due to the downward compatibility because ABAP has procedural roots and because both entire programs and reusable procedures (primarily function modules) were still supposed to be reusable with the implementation of the object-oriented programming model in the late 1990s.

1 From the technical point of view, classes are still declared and implemented in programs.
Rule

Rule 3.1: Use ABAP Objects

Use ABAP Objects wherever possible for new and further developments. Classical processing blocks may be newly created in exceptional cases only.

Details

The demand for the separation of concerns (SoC) (see Rule 2.1, Adhere to the SoC Principle) is ideally supported by using ABAP Objects as much as possible. The scope of this book doesn’t include a detailed comparison of ABAP Objects and the procedural programming model. An article in SAP Professional Journal (“Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Developer Should Give It a Second Look”) illustrated that the object-oriented programming — and here particularly ABAP Objects in comparison to classical procedural ABAP — is better suited to meet the requirements of modern programming. This article mentioned eight reasons to use ABAP Objects, which are summarized here:

1. **Data encapsulation**
   ABAP Objects enables an advanced way of data encapsulation. In classical procedural programming, the state of an application is determined by the content of the global variables of a program. In object-oriented programming, the state is encapsulated in classes or objects as instances of classes. The distribution of data across the different visibility sections of a class — public, protected, package-visible (as of Release 7.2), and private — enables a clear differentiation between externally and internally usable data. Even without an in-depth object-oriented modeling, application programs benefit from these attributes with regard to stability and maintainability.

2. **Explicit instantiation**
   ABAP Objects enables the multiple instantiation of a class via an explicit object creation using the `CREATE OBJECT` statement. Each instance of a class (object) has its own state that is determined via the values of its attributes and can be changed via the methods of the class. Automatic garbage collection ensures that objects that are no longer required are deleted from the memory. Procedural models do not provide multiple instantiation, which is why they oblige you to apply stateless functions on separately stored data.

3. **Inheritance**
   ABAP Objects enables the reuse of classes through inheritance, where classes with special behaviors are derived from more general classes, and only the dif-
ferences must be implemented anew. In the procedural model, you can use existing functions only as available, or you must create new ones.

4. **Interfaces**

In ABAP Objects, you can address objects via standalone interfaces. This way, developers don’t need to take care of the implementation details of the class behind the interface. This way, the provider of an interface can change the underlying implementations without having to modify the programs that the interface uses. The procedural model does not have such a concept of standalone interfaces.

5. **Events**

ABAP Objects makes it easier to implement event-driven program flows. A publish-and-subscribe mechanism can loosely couple applications, without the trigger of an event having to know anything about any possible handlers. This allows greater flexibility than the procedural approach, in which the programs are much stronger coupled, and the program flow is usually much more strictly defined.

6. **Explicit orthogonal concepts**

ABAP Objects contains a small number of closely defined, mutually orthogonal, fundamental concepts, which makes it more reliable and less error-prone than classical ABAP. Classical procedural ABAP is dominated by implicit behaviors in which programs are controlled by implicit events of the runtime environment and via global data. The concepts of ABAP Objects, however, are explicitly shown in a program. ABAP Objects is easier to learn and use than classical procedural ABAP.

7. **Cleansed syntax**

ABAP Objects contains cleansed syntax and semantics rules. Classical procedural ABAP is a language that has evolved over time, and contains several obsolete and overlapping concepts. With the implementation of ABAP Objects, classes and methods provided a field for cleansed syntax and semantics rules, which was completely unaffected by the requirements on the downward compatibility. This way, in ABAP Objects, that is, within classes and methods, most obsolete and error-prone language constructs were syntactically forbidden. Also, questionable and potentially faulty data accesses are checked more closely and may also be forbidden. The syntax cleansing enforces the use of the ABAP language in classes, which can only be claimed through guidelines outside of classes (see Section 3.3, Modern ABAP).
8. **Access to new technologies**

Often, ABAP Objects is the only way of using new ABAP technologies. For example, GUI controls, Web Dynpro ABAP, Runtime Type Services (RTTS), or the Internet Communication Framework (ICF) provide exclusively class-based interfaces. If programs that use such services would still be implemented purely procedurally, this would result in an unnecessary mix of programming models and a correspondingly increased complexity.

So the urgent recommendation to use ABAP Objects has aspects both in form and content:

- As detailed in reasons 1 through 5, the object-oriented programming model is better suited to keep the complexity of software manageable through principles, such as encapsulation and inheritance. Admittedly, a good object-oriented design is not an easy task, and there are still developers with little experience in this area even today. Against this background, those who still think about approaching a new development in the classical procedural way have to bring to mind that even the procedural event-driven ABAP programming model with its system events is not easy to understand.

- Reasons 6 through 8 describe rather formal aspects suggesting that you should create procedures only in the form of methods today, even if no real object-oriented design is present. Function modules and subroutines should be created only in exceptional cases in which ABAP Objects doesn't provide any alternatives yet.

In Chapter 5, Section 5.1, Object-Oriented Programming, provides notes and recommendations on the successful use of ABAP Objects.

**Exception**

In the current state (Releases 7.0 EhP2 and 7.2), the following attributes are still missing in ABAP Objects to replace classical processing blocks with methods:

- Remote Method Invocation (RMI) as a replacement for Remote Function Call (RFC)
- A replacement for the call of update function modules (CALL FUNCTION IN UPDATE TASK)
- A replacement for the call of subroutines during COMMIT WORK and ROLLBACK WORK (PERFORM ON COMMIT/ROLLBACK)
Object-oriented handling of classical dynpros, including selection screens as a replacement for dialog transactions, \texttt{CALL SCREEN} and \texttt{CALL SELECTION-SCREEN}.

Dynamic generation of classes as a replacement for the classical dynamic program generation (\texttt{GENERATE SUBROUTINE POOL}).

Direct support of background processing as a replacement for the call of executable programs (\texttt{SUBMIT VIA JOB}).

Exactly for such cases, the following classical processing blocks may still be created in new programs:

- Function modules are still required for RFC and the update and are recommended for the call of classical dynpros and selection screens (see Rule 5.19, Encapsulate Classical Dynpros and Selection Screens).

- Subroutines are still required for \texttt{PERFORM ON COMMIT/ROLLBACK} and in dynamically generated subroutine pools (\texttt{GENERATE SUBROUTINE POOL}).

- Dialog modules and event blocks for selection screen events are still required in function groups that wrap the classical dynpros and selection screens (see Rule 3.2, Select the Appropriate Program Type).

- The \texttt{START-OF-SELECTION} event block is still required in executable programs that are intended for background processing.

Within such a processing block, however, you should immediately delegate the execution to a suitable method (see Rule 6.37, No Implementations in Function Modules and Subroutines, and Rule 6.44, No Implementations in Dialog Modules and Event Blocks). This doesn’t need to be a method of a global class but can absolutely be located within the associated main program within the scope of a local class. To ensure that the system implements the same stricter check in such processing blocks as in the methods, you can activate the \texttt{OBSCOUTE STATEMENTS \{OO CONTEXT\}} check in the extended program check (see Section 3.4.2, Extended Program Check).

**Bad Example**

Listing 3.1 contains a rudimentary implementation of handling different types of bank accounts in a function group and their use in a program, whereas only the “withdrawal of an amount” function is shown. The function modules of the function group work on external data that are loaded into a global internal table in the \texttt{LOAD-OF-PROGRAM} event. The control for whether a checking account or savings
account is handled is carried out via an input parameter, and the different handling is delegated to various subroutines via a CASE WHEN control structure, whereas no reuse takes place. The subroutines access the global internal table. The withdraw function module is called for different accounts in an application program. The exception handling is carried out classically using further CASE WHEN control structures for querying sy-subrc.

FUNCTION-POOL account.

DATA account_tab TYPE SORTED TABLE OF accounts
   WITH UNIQUE KEY id.

LOAD-OF-PROGRAM.
   "fetch amount for all accounts into account_tab"
   ...
   ...

FUNCTION withdraw.
   *-----------------------------------------------------
   * IMPORTING
   *   REFERENCE(id) TYPE accounts-id
   *   REFERENCE(kind) TYPE c DEFAULT 'C'
   *   REFERENCE(amount) TYPE accounts-amount
   * EXCEPTIONS
   *      negative_amount
   *      unknown_account_type
   *-----------------------------------------------------
   CASE kind.
      WHEN 'C'.
         PERFORM withdraw_from_checking_account
               USING id amount.
      WHEN 'S'.
         PERFORM withdraw_from_savings_account
               USING id amount.
      WHEN OTHERS.
         RAISE unknown_account_type.
   ENDCASE.
ENDFUNCTION.

FORM withdraw_from_checking_account
   USING l_id   TYPE accounts-id
         l_amount TYPE accounts-amount.
   FIELD-SYMBOLS <account> TYPE accounts.
   READ TABLE account_tab ASSIGNING <account>
WITH TABLE KEY id = l_id.
ACCOUNT = ACCOUNT - L_AMOUNT.
IF ACCOUNT < 0.
    "Handle debit balance
    ...
ENDIF.
ENDFORM.

FORM withdraw_from_savings_account
    USING l_id TYPE accounts-id
    L_AMOUNT TYPE accounts-amount.
FIELD-SYMBOLS <ACCOUNT> TYPE accounts.
READ TABLE account_tab ASSIGNING <ACCOUNT>
    WITH TABLE KEY id = l_id.
IF <ACCOUNT>_WA-AMOUNT > L_AMOUNT.
    <ACCOUNT>-AMOUNT = <ACCOUNT>-AMOUNT - L_AMOUNT.
ELSE.
    RAISE negative_amount.
ENDIF.
ENDIF.
ENDFORM.

******************************************************
PROGRAM bank_application.
...
CALL FUNCTION 'WITHDRAW'
    EXPORTING
        ID = ...
        KIND = 'C'
        AMOUNT = ...
    EXCEPTIONS
        unknown_account_type = 2
        negative_amount = 4.
    CASE sy-subrc.
        WHEN 2.
            ...
        WHEN 4.
            ...
        END_CASE.
...
CALL FUNCTION 'WITHDRAW'
    EXPORTING

id                   = ...
kind                 = 'S'
amount               = ...

EXCEPTIONS
  unknown_account_type = 2
  negative_amount     = 4.

CASE    sy-subrc.
  WHEN  2.
    ...
  WHEN  4.
    ...
ENDCASE.

Listing 3.1  Modeling of Bank Accounts in Function Groups

Good Example

Listing 3.2 contains a rudimentary implementation of handling different types of
bank accounts in classes and their use in a class, whereas only the “withdrawal of
an amount” function is shown again.

The different types of accounts are implemented in subclasses of an abstract class
for accounts. Each instance of an account is provided with the required data in its
constructor. If required, the application class generates instances of accounts of the
desired type and uses their methods polymorphically via a superclass reference
variable. The exception handling is carried out via class-based exceptions. CASE
WHEN control structures are not required any more. As already announced in the
description of the examples in Chapter 2, Section 2.1, Separation of Concerns, no
overhead of code is generated here any longer for the use of classes in comparison
to the procedural programming.

CLASS  cx_negative_amount  DEFINITION PUBLIC
      INHERITING FROM  cx_static_check.
ENDCLASS.

CLASS  cl_account  DEFINITION ABSTRACT PUBLIC.
  PUBLIC SECTION.
  METHODS: constructor  IMPORTING  id  TYPE  string,
           withdraw    IMPORTING  amount  TYPE  i
             RAISING  cx_negative_amount.
  PROTECTED SECTION.
  DATA    amount  TYPE  accounts-amount.
ENDCLASS.
CLASS cl_account IMPLEMENTATION.
  METHOD constructor.
    "fetch amount for one account into attribute amount ..."
  ENDMETHOD.
  METHOD withdraw.
    me->amount = me->amount - amount.
  ENDMETHOD.
ENDCLASS.

CLASS cl_checking_account DEFINITION PUBLIC
  INHERITING FROM cl_account.
  PUBLIC SECTION.
  METHODS withdraw REDEFINITION.
ENDCLASS.

CLASS cl_checking_account IMPLEMENTATION.
  METHOD withdraw.
    super->withdraw( amount ).
    IF me->amount < 0.
      "Handle debit balance ..."
    ENDIF.
  ENDMETHOD.
ENDCLASS.

CLASS cl_savings_account DEFINITION PUBLIC
  INHERITING FROM cl_account.
  PUBLIC SECTION.
  METHODS withdraw REDEFINITION.
ENDCLASS.

CLASS cl_savings_account IMPLEMENTATION.
  METHOD withdraw.
    IF me->amount > amount.
      super->withdraw( amount ).
    ELSE.
      RAISE EXCEPTION TYPE cx_negative_amount.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

*************************************************************************

CLASS bank_application DEFINITION PUBLIC.
PUBLIC SECTION.
  CLASS-METHODS main.
ENDCLASS.

CLASS bank_application IMPLEMENTATION.
  METHOD main.
    DATA: account1 TYPE REF TO cl_account,
         account2 TYPE REF TO cl_account.

    ...  
    CREATE OBJECT account1 TYPE cl_checking_account
      EXPORTING id = ...
    CREATE OBJECT account2 TYPE cl_savings_account
      EXPORTING id = ...

    ...  
    TRY.
      account1->withdraw( ... ).
      account2->withdraw( ... ).
    CATCH cx_negative_amount.
    ...
  ENDMETHOD.
ENDCLASS.

Listing 3.2 Modeling of Bank Accounts in Classes

3.2 Program Type and Program Attributes

Already when you create an ABAP program, you determine the stability and maintainability by selecting the program type and the program attributes. Among other things, the program type and the program attributes specify the severity of the syntax check. Another important attribute of programs (like all other development objects) is their original language.
3.2.1 Program Type

Background
Each ABAP program has a program type that specifies the declarations and processing blocks a program can contain and how it can be executed via the ABAP runtime environment. The possible program types in ABAP are as follows:

- **Executable program**
  An executable program can contain all possible declarative statements. All processing blocks are possible except for function modules. It supports classical dynpros as well as selection screens and can be executed both using the `SUBMIT` statement and transaction codes. You can create an executable program using the ABAP Editor.

- **Class pool**
  A class pool always contains declarative statements for a global class and can also include declarative statements for local types, interfaces, and classes. Only methods are possible as processing blocks. It doesn’t support classical dynpros or selection screens. You can call the methods of the global class from the outside depending on the visibility and execute the public methods of the global class using transaction codes. You can create a class pool using the Class Builder.

- **Interface pool**
  An interface pool can only contain the declarative statements for a global interface. Processing blocks and classical dynpros or selection screens are not possible. You cannot call or execute an interface pool, and you create it using the Class Builder.

- **Function group (function pool)**
  A function group can contain all types of declarative statements. All processing blocks are supported except for the reporting event blocks. They support classical dynpros as well as selection screens. You can call their function modules, but you can also access the dynpro processing of the function group using transaction codes. You can create a function group using the Function Builder.

- **Module pool**
  A module pool can contain all possible declarative statements. All processing blocks are supported except for the reporting event blocks and function modules. It supports classical dynpros as well as selection screens, and you can
execute it using transaction codes. You can create a module pool using the ABAP Editor.

- **Subroutine pool**
  A subroutine pool can contain all possible declarative statements. The \texttt{LOAD-OF-PROGRAM} event block as well as subroutines and methods are possible as processing blocks. It doesn’t support classical dynpros or selection screens. You can call the subroutines, but you can also execute methods using transaction codes. You can create a subroutine pool using the ABAP Editor.

- **Type group (type pool)**
  A type group can contain the declarative statements, \texttt{TYPES} and \texttt{CONSTANTS}. Processing blocks and classical dynpros or selection screens are not possible. You cannot call or execute a type group. You create a type group using the ABAP Dictionary.

In addition to the mentioned compilation units, that is, programs that can be compiled independently, there are also \textit{include programs}, which will be discussed separately in Chapter 4, Section 4.5, Source Code Organization.

In ABAP, a program execution means that the system loads a program into the memory and executes one or more of its processing blocks. Here, you differentiate between independent and called program execution:

- **Standalone program execution**
  In the standalone program execution, you start the program either using a transaction code (\texttt{CALL TRANSACTION} and \texttt{LEAVE TO TRANSACTION} statements) or using the \texttt{SUBMIT} statement for an executable program. The \texttt{SUBMIT} statement also allows for the execution in a background process.

- **Called program execution**
  In the called program execution, a running program calls a procedure (method, function module, or subroutine) of another program that is loaded into the internal session of the caller if required (see Chapter 6, Section 6.5.6, Internal and External Procedure Calls).

The program flow within the standalone program execution depends on the selected program type and the type of the program call:

- In a program call via a transaction, you must differentiate between \textit{object-oriented (OO transaction)} and \textit{dialog transactions}. For object-oriented transactions, the transaction code is linked with a method of a local or global class. The pro-
gram flow is determined by this method. Dialog transactions, however, are linked with a classical dynpro of the program. Here, the program flow is determined by the associated dynpro flow logic.

The program flow of an executable program that was started using submit is determined by the reporting process of the ABAP runtime environment. Here, the runtime environment calls the different reporting event blocks, START-OF-SELECTION, GET and END-OF-SELECTION, of the program.

You must select the program type taking into account the technical attributes of a program mentioned here and the requirements on the program execution. Not all mentioned program types can still be used reasonably for new developments.

**Rule**

**Rule 3.2: Select the Appropriate Program Type**

To select the program type, proceed as follows:

- The program type, class pool or interface pool, automatically arises for global classes and interfaces.
- For the implementation of completed functionality that is not supposed to be displayed in the class library, you can use the program type, subroutine pool, for local classes.
- If you require function modules, this automatically results in the function group program type. Furthermore, you must use function groups to wrap classical dynpros or selection screens.
- If an execution is required within the scope of a background processing, this automatically results in the executable program type.
- No new module pools and type groups are supposed to be created any longer.

**Details**

The hierarchy provided in Rule 3.2 for selecting the program type results from the basic rule described in Section 3.1, ABAP Objects as a Programming Model, which stipulates the use of ABAP Objects. The following list further describes the specific aspects:

---

2 As of Release 7.2, you can use the operational package concept to restrict the usability of global classes to one package so that this role of subroutine pools becomes less important.
If functionality will be provided across the whole package or system in the context of ABAP Objects, this is done via the global classes or interfaces that implicitly have the program type, class pool, or interface pool. The call is done either via a method call or an OO transaction if you want a standalone program execution.

To implement closed functionality that is not supposed to be called via a method call but instead by using a transaction code, which additionally neither requires a parameter transfer nor has a user interface (UI), you can use the subroutine pool program type. The implementation will be carried out only via local classes, and the program call will be carried out via an OO transaction. Subroutine pools were — as the name suggests — originally intended for subroutines that were called from other programs. Because subroutines, and particularly their external call, are declared as obsolete within these programming guidelines, this intended use for subroutine pools is no longer given. Instead, subroutine pools are proposed as independent containers for local classes because they are hardly impacted by implicit processes of the ABAP runtime environment otherwise.

Remote-enabled Function Modules (RFM), which provide functionality via the RFC interface either across servers or across systems or are used for parallelization, can only be created in a function group. The implementation of the actual functionality, however, is to be carried out in a class, for example, in a local class within the function group (see Rule 6.37, No Implementations in Function Modules and Subroutines).

For update function modules, which are called within the update using CALL FUNCTION IN UPDATE TASK, the same applies as for RFMs.

Programs with a classical dynpro interface or selection screens (as far as they should still be required; see Rule 5.18, Use Web Dynpro ABAP) should also be created as a function group, which only implements the UI but doesn’t contain its own application logic (see Rule 2.1, Adhere to the SoC Principle, and Rule 5.19, Encapsulate Classical Dynpros and Selection Screens). This program type is suitable because it can contain both classical dynpros and an external functional interface in the form of function modules. The dialog modules of the function group called by the dynpro flow logic should basically contain method calls only, for instance, for methods of local classes.

An executable program includes several event blocks that are executed when the various reporting events occur. This form of event control is largely obsolete
and should no longer be used. Executable programs should only be used where they are technically required, thus mainly for background processing. In this case, too, the actual implementation should be carried out in methods, for example, via methods of a local class within the executable program. The event block of the initial event, **START-OF-SELECTION**, should only include a method call (see Rule 6.44, No Implementations in Dialog Modules and Event Blocks), and other event blocks should no longer occur.

- The module pool used to be the program type, which was traditionally used for the classical dialog programming with dynpros. As discussed in Chapter 2, Section 2.1, Separation of Concerns, the concept of SoC is not sufficiently supported through module pools. For this reason you should not create any new module pools. Instead, you should encapsulate classical dynpros in function groups if they still need to be used.

- The type group program type was initially implemented as a workaround because occasionally it was not possible to define any types for internal tables in the ABAP Dictionary. The same applied to the global storage of constants. Both gaps have been closed in the meantime. In the ABAP Dictionary, you can define any types, and in global classes and interfaces, it is possible to create both types and constants for package-wide or system-wide use. For this reason, the type group program type is obsolete, and no new type groups are supposed to be created any longer (in Chapter 6, see Section 6.1.2, Declaration of Data Types and Constants).

**Note**

In cases in which you still use program types other than class and interface pools, you should activate the check **OBsolete Statements (OO Context)** in the extended program check (see Section 3.4.2, Extended Program Check) to implement the same more stringent syntax check for the program components, which are not implemented in local classes, as within classes.

### 3.2.2 Program Attributes

**Background**

In addition to further, less important properties, each ABAP program has a set of program attributes that control specific aspects of the program behavior and the syntax check severity. This includes:
Unicode checks active
For creating a Unicode program.

Fixed point arithmetic
For considering the decimal separator in operations with packed numbers.

Logical database
For connecting an executable program with a logical database.

You define the program attributes when you create a program using the corresponding tool (Class Builder, Function Builder, ABAP Editor), and it is technically possible to change them later.

Rule

<table>
<thead>
<tr>
<th>Rule 3.3: Accept the Standard Settings for Program Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set the program attributes for new programs as follows:</td>
</tr>
<tr>
<td>▶ UNICODE CHECKS ACTIVE activated</td>
</tr>
<tr>
<td>▶ FIXED POINT ARITHMETIC activated</td>
</tr>
<tr>
<td>▶ No assignment to a logical database</td>
</tr>
<tr>
<td>These settings correspond to the default values when you create a new program, so you can keep them without any changes. Once the program attributes are set, you should not change them retroactively.</td>
</tr>
</tbody>
</table>

Details

Different behaviors or check severities are only provided for compatibility reasons to keep existing programs compilable and executable. New programs should not use obsolete settings by any means.

When you create a new program, the UNICODE CHECKS ACTIVE attribute is already set as a default setting. This attribute must never be reset. Only when the Unicode checks are activated can you ensure that the program can be executed both in Unicode systems and in non-Unicode systems and that it provides the same results, respectively.\(^3\) When you prepare a non-Unicode system for

---

\(^3\) A program with activated Unicode checks is referred to as a *Unicode program*. A *Unicode system* represents an SAP system in which the characters are displayed in Unicode format (ISO/IEC 10646) (currently UTF-16 with platform-dependent byte order). On a Unicode system, you can only execute Unicode programs; Unicode programs, however, can also be executed on non-Unicode systems. The programs provided by SAP are usually Unicode programs.
the migration to Unicode, all existing non-Unicode programs must be converted into Unicode programs. The activation of the Unicode checks only provides benefits for the developer, for instance, in the form of a more stringent static type check and a stricter separation of byte and character string processing.

- When you create a new program, the **Fixed Point Arithmetic** attribute is already set as a default setting. This attribute must never be reset. If the fixed point arithmetic is deactivated, the position of the decimal separator of packed numbers (type $p$) is only considered for the display in a classical dynpro or for the formatting using `WRITE TO` but not for calculations. Today, such behavior meets the developer's expectations only in rare cases. If the calculation is supposed to be carried out with packed numbers without any decimal places, this must be specified by adding `DECIMALS 0` in the declaration.

- When you create a new executable program, the **Logical Database** attribute is empty. Via this attribute, the executable programs are assigned to a logical database, which combines the selection screen and the flow of the program with the selection screen and the flow of the logical database. Logical databases should no longer be used because they are based on the cross-program use of global data, implicit subroutine calls, and the reporting event control, and are therefore contrary to modern concepts. Existing logical databases can be accessed using the `LDB_PROCESS` function module if required, which can be called from a method, for example. You should no longer create new logical databases. Instead, you should provide a corresponding service via a global class.

Because a subsequent change to the program attributes potentially involves change-over effort, you should set the correct attributes right from the start and not change them later on. Particularly for attributes that influence the syntax check (currently the Unicode check) you should always decide for the highest possible check severity to be well prepared for subsequent, possibly requested changeovers.

The following sections assume that you only work with the activated Unicode check and the fixed point arithmetic and without the logical databases. For obsolete or problematic language constructs, which are only available if the Unicode

---

4 A logical database is a special development object that is processed in the Logical Database Builder and provides other ABAP programs with data from the nodes of a hierarchical tree structure. A logical database includes a hierarchical structure, a database program written in ABAP, and its own standard selection screen.
checks are switched off, no special rule is created in these guidelines any longer. They are only mentioned briefly within the list of the obsolete language elements (see Appendix A, Obsolete Language Constructs).

**Bad Example**

Figure 3.1 shows an ABAP program in which **UNICODE CHECKS ACTIVE** is not selected in the program attributes contrary to the recommendation of Rule 3.3, **Accept the Standard Settings for Program Attributes**.

![Figure 3.1](image)

**Figure 3.1** Allowed Subfield Access to a Structure in a Non-Unicode Program

In the non-Unicode program shown in Figure 3.1, you can readily execute a write access to a subfield across two numeric components of a structure, whereas an implicit casting (see Chapter 6, Section 6.2.8, Casting) of the subarea takes place for the type c. The result in the components depends on the alignment gaps, the internal presentation of the numeric values (byte order), and the used code page, and it is therefore extremely platform-dependent. A live program must not contain such code by any means. This usually results in erroneous data or runtime errors that are difficult to reproduce.
Good Example

Figure 3.2 shows an ABAP program in which the UNICODE CHECKS ACTIVE attribute is selected in the program attributes in accordance with Rule 3.3, Accept the Standard Settings for Program Attributes.

**Figure 3.2 Syntax Error During the Subfield Access to a Structure in a Unicode Program**

In the Unicode program of Figure 3.2, the code of Figure 3.1 results in a syntax error. Unwanted subfield accesses are prohibited just like any other unwanted accesses to structures or other parts of the working memory. If it is possible to statically determine them, this results in a syntax error similar to the one in the example shown here. Otherwise, a runtime error occurs with a meaningful short dump while the program is executed.
3.2.3 Original Language

Background
When you create a new repository object (e.g., a program, a class, or a database table in the ABAP Dictionary), you must specify its original language. This is done implicitly using the current logon language. All texts of the development object, which are created during the development and can be translated, such as descriptive short and long texts, text elements of a program, and also the documentation of data types or interfaces, are assigned to the specified original language. The creation of the texts in other languages is implemented in a translation process, which is detached from development, from the original language into the target languages.

Currently, there is no technical support for the project-wide replacement of the previously selected original language with another language.

Rule

<table>
<thead>
<tr>
<th>Rule 3.4: Determine Original Language at Project Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before you start the implementation, determine a carefully selected original language for the repository objects at project level. Developers may create their development objects only in the original language determined for the respective project (or for a subproject in exceptional cases).</td>
</tr>
</tbody>
</table>

Details
When you determine the original language, you should proceed as follows:

- In case of a unilingual staff assignment of all development groups that participate in a project, the original language of all development objects is the native language of all developers involved (unilingual development).
- In case of a multilingual staff assignment of the development groups,
  - the original language of all development objects is either a language that all people involved understand, which is usually English (unilingual development),
  - or the original language of the development objects in individual parts of the project is based on the native language of the developers that mainly work on these parts (multilingual development).
Program Type and Program Attributes

Unilingual development groups normally represent the ideal case but cannot always be formed today. The two possible settings for multilingual developer groups — unilingual or multilingual development — meet two different requirements, which are contradictory:

- When a user logs on to a system in a language different from the original language, it is usually not possible to reasonably work with a product that is still being developed or newly developed until the relevant texts are available in the respective target language. The translation is usually carried out in a downstream translation system and then transported back into the development system. For this reason, an efficient development, particularly in internationally staffed development groups (that are possibly distributed across multiple locations), is only possible if a uniform original language is determined for the entire project at the beginning. This original language then enables all persons involved in the development and validation process to use the product at least for testing. In case of a unilingual development in multilingual development groups, some or even all developers of a project must create texts in a language that is not their native language.

- Usually, there is no support provided in the form of tools or defined processes for linguistic or stylistic checks of the UI texts and documentations, which developers create in languages that are not their native language. Therefore, it is desirable that the developers that participate in the development of user dialogs and documentations work in their native language and that trained translators translate these texts into their native language based on specified terminology. The latter is the reason why English is not claimed as the comprehensive uniform original language of all development projects. Instead, unilingual development groups should definitely work in their native language with a possibly downstream translation.

In multilingual development groups, it ultimately depends on the case to specify the original language for each development object. The first reason usually outweighs so that a unilingual development must be implemented in international development to use the development resources of a project as effectively as possible. In individual cases, it can absolutely make sense to specify the native language of the developers as the original language in subprojects in which a high quantity of text must be created.\(^5\)

\(^5\) This particularly concerns the SAP-internal development in which large parts are still implemented by German-speaking developers.
In multilingual projects, functions that are related to one business case should be developed in one language only, at least at package level. Table contents should also be created in a uniform language.

**Tip**
Because the original language is determined by the logon language when a repository object is created, you must deliberately decide on a logon language when creating and processing repository objects.

**Note**
Irrespective of whether you implement a unilingual or multilingual development with a project, you must always create a uniform *terminology* for all texts created in the project before you start a development and adhere to it consistently. In a multilingual development, you should implement the translation of the terminology into the languages used, preferably before the development starts so that the developers can use it. In addition, you must always comply with the existing standards for UI texts and documentation (in Chapter 2, see Section 2.3, Correctness and Quality).

## 3.3 Modern ABAP

**Background**
ABAP is a living programming language that is continually being developed. Since the implementation of ABAP some 30 years ago, new ABAP programs are continuously being developed, while the ABAP language itself is advanced at the same time. Further developments of the ABAP language are either extensions of the existing language attributes to implement new functionality or to replace existing functionality with more advanced concepts. The replacement of existing language elements with new ones usually makes the existing language elements superfluous or obsolete. The most prominent example of a further development of the ABAP language is still the implementation of ABAP Objects for Release 4.6.

With regard to the ABAP language, SAP has committed itself to a policy of strict downward compatibility. This means, on the one hand, that an ABAP program written for SAP R/3 Release 3.0 can be executed on SAP NetWeaver AS ABAP in Release 7.2 without any modifications, at least as long as it is a non-Unicode system. On the other hand, this also means
An experienced developer has hardly ever been forced to break with old habits and to engage in new concepts. The only exception is the changeover to Unicode systems for which the ABAP programs must be converted into Unicode programs with slightly changed syntax rules.

ABAP beginners get confused by the multitude of options available to do one and the same thing. If in doubt, older programs serve as templates, and the obsolete concepts are frequently used instead of the new ones.

To remedy these problems, you can use the following simple rule.

**Rule**

**Rule 3.5: Do Not Use Obsolete Language Elements**

Do not use obsolete language elements for new developments. It is also recommended to incrementally change over to newer concepts as they are available.

**Details**

Newer language elements are always the better language elements. Obsolete language elements are only provided for downward compatibility reasons. A statement or statement addition is declared as obsolete only when a more powerful alternative exists or when the language element is determined as prone to errors (in the sense that it invites insecure and nonrobust programming). For this reason, secure and robust programming is not possible if obsolete language elements are used, which makes the use of such obsolete language elements out of the question for new developments.

If ABAP Objects is used, the majority of the obsolete statements and additions are already prohibited syntactically. For this reason among others, it is strongly recommended to use ABAP Objects (see Rule 3.1, Use ABAP Objects). Outside of ABAP Objects, that is, in cases that are still allowed according to Section 3.1, ABAP Objects as a Programming Model, you must make sure no obsolete language elements are used. For this purpose, Appendix A, Obsolete Language Constructs, provides an overview of the obsolete statements and statement additions.

**Bad Example**

Listing 3.3 shows the solution of a task using obsolete language elements. A procedure is supposed to replace all occurrences of a substring in a text with a new character string new if the substring is not at the end of a word.
FORM bad_example USING substring TYPE csequence
               new   TYPE csequence
               CHANGING text   TYPE csequence.
DATA: pattern TYPE string,
       subrc  TYPE sy-subrc.
CONCATENATE '**' substring INTO pattern.
SEARCH text FOR pattern.
IF sy-subrc <> 0.
   CLEAR subrc.
   WHILE subrc = 0.
      REPLACE substring WITH new INTO text.
      subrc = sy-subrc.
   ENDWHILE.
ENDIF.
ENDFORM.

Listing 3.3 Use of Obsolete Language Elements

In Listing 3.3, aside from the modularization with FORM-ENDFORM, the statement
SEARCH and the used variant of REPLACE are obsolete as of Release 7.0. Furthermore, as of Releases 7.0 EhP2 and 7.2, a character string operator && is available
as a replacement for CONCATENATE.

Good Example

Listing 3.4 executes the same task as Listing 3.3; however, it uses the latest available language elements.

METHOD good_example.
   FIND REGEX substring && '\b' IN text.
   IF sy-subrc <> 0.
      REPLACE ALL OCCURRENCES OF substring IN text WITH new.
   ENDIF.
ENDMETHOD.

Listing 3.4 Use of Modern Language Elements

The subroutine is replaced with a method. By using FIND in connection with a
regular expression, which is composed using the character string operator &&, you
no longer require any helper variable. The WHILE loop is replaced with REPLACE
ALL OCCURRENCES, whereas another helper variable is omitted, and the control flow
is moved to the ABAP runtime environment. The latter increases the execution
speed and is also helpful to meet Rule 4.22, Restrict the Nesting Depth of Control Structures, on limiting the maximum nesting depth.

**Note**

In connection with Rule 3.5, Do Not Use Obsolete Language Elements, the question on the coexistence of old and new concepts within a program unit arises. There is only one area in which this is clearly syntactically defined, that is, the use of the classical and the class-based exception concept (in Chapter 5, see Section 5.2.2, Classical and Class-Based Exceptions) in processing blocks. Else, obsolete language elements can be directly next to new language elements in a program part. In this context, it is recommended to design the use as consistently as possible within a context, that is, to not use different statements, such as `FIND` and `SEARCH`, in parallel for the same purpose.

However, this does not mean that if already existing procedures are extended, you should still use obsolete language elements for consistency reasons just because they already exist there. You should rather use the opportunity and directly change the entire procedure to the corresponding new language elements. By covering the procedures to be changed with module tests, you can ensure that no surprises occur during such a changeover.

### 3.4 Checks for Correctness

In Chapter 2, Section 2.3, Correctness and Quality, we already discussed the correctness and quality of programs in general and briefly presented the tools that are available for their check. This section now particularly discusses the syntactic correctness of ABAP programs, which is controlled using the syntax check and the extended program check as well as the standard check of the Code Inspector and the new ABAP Test Cockpit (ATC).

#### 3.4.1 Syntax Check

**Background**

The syntax check provides syntax errors and syntax warnings:

- As soon as a syntax error occurs, the system ends the check and displays a corresponding error message. In many cases, the system proposes a correction that
you can accept. Programs with syntax errors can be activated, but they cannot be generated and therefore executed. In the extended program check, the syntax errors are reported as fatal errors. Syntax errors must be remedied by all means.

- If a syntax warning occurs, the syntax check is not terminated, and the program can be executed in principle. The syntax warnings are displayed in the ABAP Editor after an execution of the syntax check and the extended program check (see Section 3.4.2, Extended Program Check). When a program is activated, the system only generates syntax warnings if there are syntax errors at the same time.

The warnings notified by the syntax check are subdivided into three priorities that are only displayed by the extended program check, however:

- **Priority 1**
  Errors that apparently will lead to a program termination if the ABAP program is executed. This priority also contains all constructs that should not be used at all because they suggest program errors and very likely lead to incorrect behavior.

- **Priority 2**
  This priority refers to all constructs that do not necessarily lead to error behavior but are obsolete and supposed to be replaced with current constructs, for example. Errors of priority 2 can become errors of priority 1 or syntax errors in future releases.

- **Priority 3**
  Summarizes all errors whose correction is desirable, but not necessarily required for the current release. However, a tightening of the priority in future releases is not excluded.

The severity of the ABAP syntax check is determined by the decisions that were made when the program was created (see Section 3.2, Program Type and Program Attributes). As a result, program constructs that only lead to syntax warnings outside of classes or in non-Unicode programs can represent real syntax errors within

---

6 Of course, testing tools that include the checks of the extended program check, such as the Code Inspector and the SAP-internal ABAP Test Cockpit (as of Releases 7.0 EhP2 and 7.2), also display syntax warnings.
classes or in Unicode programs. As of Releases 7.0 EhP2 and 7.2, you can suppress selected syntax warnings using pragmas.7

With the implementation of the operational package concept as of Release 7.2, the syntax check also checks package violations. Here, it depends on the encapsulation level that was set for the corresponding package whether a syntax error or only a syntax warning occurs.

**Rule**

**Rule 3.6: Consider Syntax Warnings**

Take all warnings of the ABAP syntax check seriously. No syntax warnings may occur in a completed program.

**Details**

You must always correct the causes of syntax warnings because they generally lead to unpredictable errors. Such warnings are frequently promoted to errors by SAP in a subsequent release of SAP NetWeaver AS ABAP. In this case, a program that initially only included syntax warnings is syntactically incorrect and can no longer be used after an upgrade. The same applies to the changeover from non-Unicode programs to Unicode programs or to the migration of older program parts to ABAP Objects.

With regard to the package check, the consistent use of the package concept that is already available before Release 7.2 (select Package Check as Server in the Package Builder) or the specification of a weak encapsulation as of Release 7.2 represents an initial step on the way to real encapsulation. It enables the users of development objects to adapt their usage locations even before hard syntax errors occur. For this reason, particularly all warnings of the package check both before and after Release 7.2 must be taken seriously and be corrected, so that the program remains syntactically correct even after an intensified encapsulation of the used packages.

---

7 A pragma is a program directive that doesn’t influence the program flow but impacts certain checks.
Bad Example

Figure 3.3 shows a section of a non-Unicode program in a non-Unicode system in which a `VALUE` specification leads to a syntax warning because an improper initial value is set for a structure.

![Program with Syntax Warning](image)

Figure 3.3  Program with Syntax Warning

Note

In a Unicode program, that is, a program for which the `UNICODE CHECKS ACTIVE` program attribute is set, the statement that solely leads to a warning in Figure 3.3 results in a syntax error.

Good Example

Figure 3.4 shows the corrected program of Figure 3.3. The components of the structure are provided with type-specific initial values in the instance constructor. The program has no syntax warnings and is correct also as a Unicode program.
Background

You can call the extended program check for activated programs either from the ABAP Workbench or using Transaction SLIN. It performs static checks that are too involved for a normal syntax check. You can execute either individual or multiple partial tests or a standard check that includes the most critical partial tests.

The extended program check outputs errors, warnings, and messages. The standard check notifies you of the errors and warnings that are particularly critical.\(^8\) Furthermore, the extended program check also displays the errors and warnings of the syntax check.

Since Releases 7.0 EhP2 and 7.2, in the initial screen of the extended program check, you can also select a check of PROGRAMMING GUIDELINES that checks the adherence to some of the rules presented in this book that can be verified statically.

---

\(^8\) The classification of the individual results as error, warning, or message may vary depending on whether you execute a standard check or explicitly selected individual checks.
The messages of the extended program check, which are inapplicable in some special cases, can be hidden using pseudo comments and, since Releases 7.0 EhP2 and 7.2, using pragmas. Messages that directly arise from a normal syntax check could not be hidden before Releases 7.0 EhP2 and 7.2.

**Rule**

<table>
<thead>
<tr>
<th>Rule 3.7: Use the Extended Program Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the extended program check, and take its results seriously. No messages of the standard check may occur for a completed program.</td>
</tr>
</tbody>
</table>

**Details**

The errors, warnings, and messages output by the extended program check are as important as the syntax errors and syntax warnings of the syntax check (see Section 3.4.1, Syntax Check). For example, an error notified by the extended program check can indicate that a program will definitely lead to a runtime error when it is executed. Warnings and messages usually refer to a questionable use of language elements, which most likely result in unexpected program behavior.

In rare cases in which a check result provided by the extended program check is not justified, this must be documented using an appropriate pseudo comment or a pragma (since Releases 7.0 EhP2 and 7.2). The appropriate pseudo comment or the pragma is indicated in the message. This way, the system suppresses the message of the extended program check. Ideally, in less obvious situations, an additional comment should describe why the message is not applicable here.

**Tip**

The extended program check is a valuable help for writing correct ABAP programs. This advantage must not be undone by using unspecific pseudo comments or pragmas. You should never use the

```plaintext
SET EXTENDED CHECK OFF.
```

statement, which suppresses all messages of the extended program check for an entire source code section.

If the ABAP program is submitted to a code review, the results of the extended program check should be used to evaluate the quality.
Bad Example

Figure 3.5 shows the result of a partial check of the extended program check, which is implemented since Releases 7.0 EhP2 and 7.2. It indicates a particularly questionable query of the content of `sy-subrc` (in Chapter 6, see Section 6.3.4, Return Value).

![Warning of the Extended Program Check](image-url)

The program section shows a typical error in a syntactically correct program. The developer wrongly assumes that the static form of the `ASSIGN` statement sets the `sy-subrc` system field, which is not the case. This entails that the developer falsely believes that he secured his program and that an incorrect program behavior occurs if `sy-subrc` has a value that is unequal zero due to previous statements. Therefore, the big advantage of the extended program check is that the system examines not only individual statements for syntactic correctness but entire program sections for semantic errors.

Good Example

Figure 3.6 shows the corrected version of the program of Figure 3.5. Instead of the wrong query of `sy-subrc`, the logical expression `IS ASSIGNED`, as recommended in the documentation, is used. The message of the extended program check could
also be hidden using a pseudo comment, "#EC RC_READ", or a pragma, "##SUBRC_READ" (as of Releases 7.0 EhP2 and 7.2), but this is not recommended in this case because the extended program check indicates a real problem.

![Figure 3.6 Extended Program Check Without Message](image)

### 3.4.3 Code Inspector

#### Background

The Code Inspector is a tool to statically check repository objects in terms of performance, security, syntax, and adherence to naming conventions. You can use the full functional scope of the Code Inspector using Transaction SCI to implement complex static checks as well as regular mass tests for large quantities of development objects.
You can also call the Code Inspector from the ABAP Workbench to carry out a standard set of checks for your current object, for instance, using the Program • Check • Code Inspector menu path of the ABAP Editor. The default check variant used here contains most of the checks of the extended program check (see Section 3.4.2, Extended Program Check) as well as some additional security and performance checks. Beyond that, you can integrate the Code Inspector in the release of transports.

Like in the extended program check, the results of the Code Inspector are subdivided into the three categories of error, warnings, and simple messages and can be hidden using special pseudo comments.

**Rule**

<table>
<thead>
<tr>
<th>Rule 3.8: Use Default Check Variant of the Code Inspector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute the default check variant of the Code Inspector prior to releasing a program, and remove all error messages.</td>
</tr>
</tbody>
</table>

**Details**

If you observe Rule 3.7, Use the Extended Program Check, the default check variant of the Code Inspector only indicates messages of checks that go beyond the extended program check. These messages basically concern possible performance or security risks in programs. Examples are messages on unfavorable WHERE conditions in SELECT, the pass by value instead of the pass by reference, or insecure program calls.

Compared to the messages of the extended program check, the cause of these problems cannot be corrected always so easily, for example, because there is no other option for a selection or because the clarity or robustness of a construct is considered as more important than a possible minor performance loss.

In such cases, you can suppress the messages using the appropriate pseudo comments. Such a pseudo comment clearly indicates for the reader of the program that the program author executed corresponding checks and that he suppressed the message intentionally and for good reasons. The latter can be further confirmed using additional normal comments if required (see Chapter 4, Section 4.3, Comments).
Bad Example

Figure 3.7 shows the result of a Code Inspector run for a sample class. Warnings are issued because an internal table is passed by value, and an inner join is used for database tables with activated SAP buffering in the SELECT statement.

Figure 3.7  Warnings of the Code Inspector
Good Example

Figure 3.8 shows the corrected version of the program of Figure 3.7 in which the Code Inspector no longer issues any messages.

![Image of ABAP Editor: Change Report Z_CODE_INSPECTOR]

Figure 3.8 Code Inspection Without Messages

The pass by value of the internal tables was replaced with a pass by reference. For passing the elementary parameter \texttt{langu}, the pass by value was kept for reasons of robustness. In the standard check used here, it didn't generate any warning anyhow. If the Code Inspector displays a warning in such as case, it can be hidden using the pseudo comment, \texttt{"#EC CI\_VALPAR."}
The inner join of the `SELECT` statement bypasses the SAP buffering, which would result in performance problems if the method was called frequently. But if you assume for the example displayed here that the method is part of a larger application, in which a buffering of the selected data is ensured via Shared Objects, you should preferably use the inner join instead of other constructs with lower performance, such as a nested `SELECT` loop. Therefore, the warning of the Code Inspector is hidden using the pseudo comment "#EC CI_BUFFJOIN", and the reasons are described in a normal comment.

### 3.4.4 ABAP Test Cockpit

**Background**

As of Releases 7.0 EhP2 and 7.2, the ABAP Test Cockpit (ATC) is integrated with the ABAP Workbench for internal SAP use. This framework considerably facilitates the development-oriented handling of the required tests. The ATC allows for the execution and results display of different tests for development objects, such as

- Extended program checks
- Static performance tests
- Module tests with ABAP Unit Browser
- Static usability tests
- Package checks

While the Code Inspector is only integrated with the development environment via the standard check, which is detailed in Section 3.4.3, and can otherwise only be operated using a dedicated transaction, the ATC is completely integrated with the Object Navigator and the Transport Organizer and is readily available for tests during development. After development and transport, quality managers can use the ATC to implement mass tests. For now, however, the ATC is only available at SAP and possibly at SAP partners for the development of SAP programs.

**Rule**

<table>
<thead>
<tr>
<th>✓</th>
<th>Rule 3.9: Configure and Use the ABAP Test Cockpit Correctly</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the ATC is available in your system, make sure that an ATC run for all development objects involved no longer displays any messages before a transport is released. For this purpose, you should integrate the ATC check directly into the release of transports.</td>
<td></td>
</tr>
</tbody>
</table>
Details

The ATC is the first tool that can be used both by SAP developers and within the framework of a central quality assurance. For example, if a developer checks all development objects of a package in the development system using the same ATC configuration that the quality manager uses within the framework of a mass run in a consolidation system, he can directly avoid all messages without having to wait for the quality manager's feedback.

If the ATC is available and configured correctly, Rule 3.9 includes the preceding Rules 3.6, 3.7, and 3.8.

Exception

At the moment, the ATC is not yet available for developments in customer systems.

Bad Example

Figure 3.9 shows the result of an ATC run implemented in the Transport Organizer. The checked transport request still contains erroneous objects.

Figure 3.9  Warning of the ABAP Test Cockpit
**Good Example**

Figure 3.10 shows the result of an ATC run in the Transport Organizer after the error shown in Figure 3.9 has been remedied. The transport can be released now.

![ABAP Test Cockpit Without Messages](image)

*Figure 3.10  ABAP Test Cockpit Without Messages*
Index

4GL language, background, 158
?=, notation, 143
<, notation, 135
<=, notation, 135
>=, notation, 135
=>, notation, 135
==, notation, 135
>, notation, 135
>=, notation, 135
=<, obsolete, 349
=>, obsolete, 349
<, obsolete, 349
&& -> see chaining operator, 90
=> -> see Class component selector, 109
-> -> see Instance component selector, 109
& -> see literal operator, 90
?TO, notation, 143

A

abap_bool
  background, 239
  use, 240
ABAP Database Connectivity -> see ADBC, 210
ABAP Dictionary
  Background, 220
  use, 221
abap_false
  background, 239
  use, 240
ABAP language element
  Background, 62
  Coexistence, 65
  Example, 161
  Use, 159
ABAP Objects
  Background, 41
  Example, 48
ABAP programming
  Classical, 25
  Object-oriented, 155
ABAP statement
  Arrangement, 83
  Example, 84, 85
  Notation, 135

ABAP Test Cockpit -> see ATC; ABAP Test Cockpit, 76
abap_true
  background, 239
  use, 240
abap_undefined
  use, 240
ABAP Unit
  Module test, 37
ABAP word
  Naming, 107
Abbreviated form (data cluster)
  Obsolete, 360, 361
Abbreviated form (internal table)
  Obsolete, 355
Abbreviated form (Open SQL)
  Obsolete, 360
Accessibility
  Product standard, 35, 207
Activatable assertion
  Use, 191
ADBC, 210
  Background, 210
  Use, 211
ADD
  Use, 143
ADD-CORRESPONDING
  Obsolete, 352
ADD FROM
  Obsolete, 353
Additional program group
  background, 299
ADD THEN
  Obsolete, 353
ALV, 204
  Background, 204
ALV list
  Example, 206
Anonymous container
  background, 259
Anonymous data object
  Background, 217
  dynamic memory object, 315
  example, 328
Index

Application buffer
  Background, 213
  Example, 215
  Use, 214
Application logic
  Separation of concerns, 29
Arithmetic expression
  Example, 148
  Notation, 145
ASCII character set
  use, 335
ASSERT
  Background, 190
  Use, 191
Assertion
  Background, 173, 190
  Example, 191
  Use, 173, 194
ASSIGN
  dynamic access, 321
  use, 322
ASSIGN DECIMALS
  Obsolete, 352
ASSIGNING
  dynamic access, 321
  example, 286
  use, 285
ASSIGN LOCAL COPY OF
  Obsolete, 347
Assignment
  background, 241
  conversion rule, 245
  loss-free, 246
  Notation, 143
ASSIGN TABLE FIELD
  Obsolete, 352
ASSIGN TYPE
  Obsolete, 352
ATC
  Background, 76
  Example, 77
  Use, 76
AT LINE-SELECTION
  background, 304
Atomization
  Example, 160
AT PFnn
  Obsolete, 358
AT SELECTION-SCREEN
  background, 304
  use, 305
AT USER-COMMAND
  background, 304

B

Background, 94
Background processing
  Use, 45
BAdI
  Naming convention, 103
BETWEEN
  Notation, 135
Binary floating point number
  background, 250
  example, 253
  use, 251
Bit expression
  Notation, 145
Blank lines
  Use, 87
BOM
  OPEN DATASET, 337
  text file, 337
Boolean data type
  background, 239
Bound data type
  Background, 218
  Example, 219
Boxed component, 227
  Background, 227
  use, 317
Browser Control
  Use, 208
BSP
  Use, 196
Business Add-In -> see BAdI, 103
Business Server Pages -> see BSP, 196
Byte string
  background, 236

C

Calculation
  background, 241
  Notation, 143
Calculation expression
  Notation, 145
Calculation type

background, 241

CALL CUSTOMER-FUNCTION

Obsolete, 347

CALL CUSTOMER SUBSCREEN

Obsolete, 357

CALL DIALOG

Obsolete, 347

CALL FUNCTION IN UPDATE TASK

Use, 44

CALL METHOD

Rule, 141, 142

CALL SCREEN

Background, 195

Use, 45

CALL SELECTION-SCREEN

Background, 195

Use, 45

CALL TRANSACTION

Program execution, 52

CALL TRANSFORMATION

use, 260

Camel case style

Example, 82

Use, 81

CASE

Complexity, 149

Obsolete, 349

Case sensitivity

Background, 80

Example, 82

Use, 81, 88

Casting

background, 255

CASTING

ASSIGN, 256

eexample, 257

implicit, 256

use, 322

CATCH

Background, 185

Catchable runtime error

Background, 188

Obsolete, 348

CATCH SYSTEM-EXCEPTIONS

Background, 188

Example, 189

Obsolete, 348

Use, 189

Chained statement

Background, 137

Use, 137

Chaining operator &

Use, 90

CHANGING

background, 289

extample, 291

Naming convention, 107

Use, 290

Character literal

background, 234

translatability, 329

use, 235, 330

Character literal template

use, 330

Character set

background, 335

Character string expression

Notation, 145

Character string template

use, 254

CHECK

background, 302

example, 303

Use, 302

Checkpoint group

Assertion, 190, 191

Class

Complexity, 151

Naming, 96

Class-based exception

Background, 175

Example, 177

Forwarding, 185

Handling, 185

Use, 176

Class component

Naming, 110

Class component selector

Naming, 109, 110

Classical exception

Background, 174

Example, 177

Use, 175

Classical list

Background, 204

Example, 205

Use, 205

Class pool

Local declaration, 169
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program type</td>
<td>51</td>
</tr>
<tr>
<td>Rule</td>
<td>53</td>
</tr>
<tr>
<td>Use</td>
<td>54</td>
</tr>
<tr>
<td>Class reference variable</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>167</td>
</tr>
<tr>
<td>Use</td>
<td>167</td>
</tr>
<tr>
<td>Class size</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>151</td>
</tr>
<tr>
<td>CLEANUP</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>186</td>
</tr>
<tr>
<td>Example</td>
<td>187</td>
</tr>
<tr>
<td>Use</td>
<td>187</td>
</tr>
<tr>
<td>CLEAR</td>
<td></td>
</tr>
<tr>
<td>use</td>
<td>316</td>
</tr>
<tr>
<td>CLEAR WITH NULL</td>
<td></td>
</tr>
<tr>
<td>Obsolete</td>
<td>351</td>
</tr>
<tr>
<td>Client handling</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>211</td>
</tr>
<tr>
<td>Example</td>
<td>212, 213</td>
</tr>
<tr>
<td>CLIENT SPECIFIED</td>
<td></td>
</tr>
<tr>
<td>Use</td>
<td>212</td>
</tr>
<tr>
<td>CL_JAVA_SCRIPT</td>
<td></td>
</tr>
<tr>
<td>CLASS DEFINITION</td>
<td>171</td>
</tr>
<tr>
<td>Obsolete</td>
<td>363</td>
</tr>
<tr>
<td>Code Inspector</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>72</td>
</tr>
<tr>
<td>Error</td>
<td>73</td>
</tr>
<tr>
<td>Example</td>
<td>74</td>
</tr>
<tr>
<td>Message</td>
<td>73</td>
</tr>
<tr>
<td>Naming convention</td>
<td>365</td>
</tr>
<tr>
<td>Pseudo comment</td>
<td>73</td>
</tr>
<tr>
<td>Use</td>
<td>73</td>
</tr>
<tr>
<td>Warning</td>
<td>73</td>
</tr>
<tr>
<td>Codepage</td>
<td></td>
</tr>
<tr>
<td>text file</td>
<td>337</td>
</tr>
<tr>
<td>COLLECT</td>
<td></td>
</tr>
<tr>
<td>background</td>
<td>283</td>
</tr>
<tr>
<td>use</td>
<td>284</td>
</tr>
<tr>
<td>Colon-comma logic-&gt; see chained statement</td>
<td>137</td>
</tr>
<tr>
<td>Combining characters</td>
<td></td>
</tr>
<tr>
<td>background</td>
<td>336</td>
</tr>
<tr>
<td>Comment</td>
<td></td>
</tr>
<tr>
<td>Arrangement</td>
<td>120</td>
</tr>
<tr>
<td>Background</td>
<td>115</td>
</tr>
<tr>
<td>Example</td>
<td>119, 122</td>
</tr>
<tr>
<td>Pseudo comment</td>
<td>121</td>
</tr>
<tr>
<td>translatability</td>
<td>330</td>
</tr>
<tr>
<td>Use</td>
<td>117, 148</td>
</tr>
<tr>
<td>Comment language</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>115</td>
</tr>
<tr>
<td>Example</td>
<td>116, 117</td>
</tr>
<tr>
<td>Comment line</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>115</td>
</tr>
<tr>
<td>Commercial notation</td>
<td></td>
</tr>
<tr>
<td>background</td>
<td>248</td>
</tr>
<tr>
<td>COMMON PART</td>
<td></td>
</tr>
<tr>
<td>Obsolete</td>
<td>343</td>
</tr>
<tr>
<td>COMMUNICATION</td>
<td></td>
</tr>
<tr>
<td>Obsolete</td>
<td>363</td>
</tr>
<tr>
<td>Compilation unit</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>131</td>
</tr>
<tr>
<td>Complete typing</td>
<td></td>
</tr>
<tr>
<td>background</td>
<td>296</td>
</tr>
<tr>
<td>example</td>
<td>298</td>
</tr>
<tr>
<td>Complexity</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>146</td>
</tr>
<tr>
<td>Class</td>
<td>151</td>
</tr>
<tr>
<td>Control structure</td>
<td></td>
</tr>
<tr>
<td>149</td>
<td></td>
</tr>
<tr>
<td>Dead code</td>
<td>153</td>
</tr>
<tr>
<td>Function group</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>150</td>
</tr>
<tr>
<td>COMPUTE</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>146</td>
</tr>
<tr>
<td>COMPUTE EXACT</td>
<td></td>
</tr>
<tr>
<td>use</td>
<td>146</td>
</tr>
<tr>
<td>Constant</td>
<td></td>
</tr>
<tr>
<td>Background</td>
<td>220</td>
</tr>
<tr>
<td>example</td>
<td>223, 235</td>
</tr>
<tr>
<td>Naming</td>
<td>96</td>
</tr>
<tr>
<td>use</td>
<td>234</td>
</tr>
<tr>
<td>CONSTANTS</td>
<td></td>
</tr>
<tr>
<td>background</td>
<td>220</td>
</tr>
<tr>
<td>constructor</td>
<td></td>
</tr>
<tr>
<td>METHODS, 171</td>
<td></td>
</tr>
<tr>
<td>Context</td>
<td></td>
</tr>
<tr>
<td>Naming convention</td>
<td>111</td>
</tr>
<tr>
<td>CONTEXTS</td>
<td></td>
</tr>
<tr>
<td>Obsolete</td>
<td>362</td>
</tr>
<tr>
<td>CONTROLS</td>
<td></td>
</tr>
<tr>
<td>use</td>
<td>225</td>
</tr>
<tr>
<td>Control structure</td>
<td></td>
</tr>
<tr>
<td>Chained statement</td>
<td>139</td>
</tr>
<tr>
<td>Comment</td>
<td>120</td>
</tr>
<tr>
<td>Complexity</td>
<td>149</td>
</tr>
<tr>
<td>Conversion</td>
<td></td>
</tr>
<tr>
<td>background</td>
<td>241</td>
</tr>
<tr>
<td>example</td>
<td>242</td>
</tr>
</tbody>
</table>
Conversion rule
  background, 245
CONVERT DATE
  Obsolete, 354
  use, 244
CONVERT TIME STAMP
  use, 244
Coverage Analyzer
  Test coverage, 37
CREATE addition
  CLASS DEFINITION, 171
CREATE AREA HANDLE
  Use, 214
CREATE DATA
  dynamic access, 321
  rule, 326
Customer namespace
  Name, 102
CX_DYNAMIC_CHECK
  Background, 179
  rule, 186
CX_NO_CHECK
  Background, 179
  rule, 186
CX_ROOT
  Rule, 186
CX_STATIC_CHECK
  Background, 178
  Rule, 186

D
DATA
  use, 225
Database access
  Background, 210
Database table
  Background, 208
  Use, 209
Data cluster
  Use, 209
DATA COMMON PART
  interface work area, 300
Data encapsulation
  ABAP Objects, 42
Data object
  Background, 217
  reference to, 230
Data reference
  dynamic access, 321
  example, 323
  rule, 322
  use, 323
Data type
  background, 217, 220, 228
  example, 229, 271
  Naming, 96
  Naming convention, 111
  reference to, 230
  rule, 228
Date field
  background, 243
  use, 244
Dead code
  Background, 153
  Complexity, 153
Decimal floating point number
  background, 250
  example, 253
  use, 251
Declaration
  Chained statements, 138
Default check variant
  Code Inspector, 73
DELETE dbtab
  Obsolete, 362
DEMAND
  Obsolete, 362
Dereferencing
  use, 323
DESCRIBE FIELD
  use, 325
DETAIL
  Obsolete, 358
Development language
  Background, 60
Dialog message
  Use, 193
Dialog module
  background, 288, 304
  Use, 45, 200
Dialog program
  Example, 202
  Separation of concerns, 23
Dialog transaction
  Program execution, 52
  Use, 45
Index

DIVIDE
Use, 143
DIVIDE-CORRESPONDING
Obsolete, 352
DO
Complexity, 149
Documentation
Means, 37
Product standard, 35
DO VARYING
Obsolete, 350
Downward compatibility
Background, 62
Dynamic access
background, 309, 321, 324
rule, 322
use, 311, 322
Dynamic data object
background, 313
rule, 313
use, 311
Dynamic invoke
background, 309, 324
use, 311
Dynamic memory object
administration, 318
background, 309
example, 317, 318
fill level, 320
memory consumption, 315
Dynamic programming
background, 309
example, 312
use, 310
Dynamic token specification
background, 324
example, 328
use, 311, 325
Dynpro
Background, 195
Example, 197, 202
use, 300
Use, 197

Obsolete, 354
Encapsulation
Background, 156
ENCODING
OPEN DATASET, 337
END-OF-SELECTION
background, 304
Program execution, 53
English
Comment language, 115
Development language, 61
Naming language, 92
Use, 93
EQ
Notation, 135
Error
Code Inspector, 73
Extended program check, 69
erro_message
Use, 193
Error message
Use, 193
Error situation
Handling, 172
Event
ABAP Objects, 43
Naming, 96
Event block
background, 288, 304
Use, 45
Event handler
Naming, 97
EXACT
COMPUTE, 145
MOVE, 246
Exception
Background, 172
Naming, 97
Use, 173
Exception category
Background, 178
Exception class
Background, 183
Example, 184
Use, 183
Exception handling
Background, 174
Remote Function Call, 177
Exception text

E

ecATT
Scenario test, 37
EDITOR-CALL
Index

Background, 180
Example, 182
Exclusive buffer
   Use, 214
Executable program
   Program type, 51
   Rule, 53
   Separation of concerns, 24
   Use, 54
EXIT
   background, 302
   use, 302
Exit message
   Use, 194
EXPORT
   Notation, 136
EXPORTING
   background, 289
   example, 291, 295
   Naming convention, 107
   pass by reference, 294
   rule, 294
   use, 290, 294
EXPORT TO DATA BUFFER
   use, 260, 314
EXPORT TO INTERNAL TABLE
   use, 314
Expression
   Complexity, 147
Extended program check
   Background, 69
   Error, 69
   Example, 71
   Message, 69
   Use, 70
   Warning, 69
External procedure call
   background, 298

FIELD (dynpro)
   Obsolete, 357
FIELDS
   Obsolete, 346
Field symbol
   dynamic access, 321
   example, 324
   rule, 322
typing, 297
use, 322
FIELD-SYMBOLS
   declaration, 225
   Obsolete, 344
File
   Background, 208
   Use, 209
Final class
   Use, 166
FIND
   Notation, 136
Fixed point arithmetic
   Program attribute, 56
   Rule, 56
Fixed Point Arithmetic
   Use, 57
Floating point number
   background, 250
FORM
   Obsolete, 342
Formal parameter
   rule, 290, 292
   transfer, 292
   type, 289
typing, 296
FORMAT
   Background, 195
FREE
   use, 317
Functional correctness
   Product standard, 35
Functional method
   Naming, 97, 110
Function group
   Complexity, 152
   example, 226
   Example, 45
   Global declaration part, 127
   Program type, 51
   Rule, 53
   Use, 54, 200
Function module
   background, 287, 288
   use, 300
   Use, 45
Function pool
   Program type, 51
Index

**G**

Garbage Collector
  use, 317
GE
  Notation, 135
GENERATE SUBROUTINE POOL
  background, 325
  Use, 45
Generic data type
  background, 296
Generic programming
  background, 309, 324
  use, 312
Generic typing
  background, 296
  example, 297
GET
  background, 304
  Program execution, 53
GET REFERENCE
  dynamic access, 321
Global class
  Local declaration, 169
  Naming convention, 103
  use, 221
Global data object
  background, 260
  Naming, 110
  Naming convention, 107
Global declaration
  Arrangement, 124
  Example, 126
  Top include, 125
Global declaration part
  Background, 124
Global exception class
  Naming convention, 103
Global interface
  Naming convention, 103
Globalization
  product standard, 329
  Product standard, 35
Global variable
  use, 224
GT
  Notation, 135
GUI status
  use, 300

**GUI title**
  translatability, 332

**H**

Hashed table
  background, 273
  table type, 274
  use, 276
Hash key
  background, 276
  use, 278
Header
  dynamic memory object, 318
Header comments
  Use, 118
HEADER LINE
  Obsolete, 345
Helper variable
  declaration, 230
  example, 231
  Example, 148
  Use, 147

**I**

IF
  Complexity, 149
IMPORT
  Notation, 136
IMPORTING
  background, 289
  Naming convention, 107
  use, 290
IN
  Notation, 135
Include program
  Background, 130, 132
  Example, 134
  Multiple use, 133
  use, 222
  Use, 131
INCLUDE STRUCTURE
  background, 226
  use, 227
INCLUDE TYPE
  background, 226
  example, 227
  use, 227
Index

Indentation
  Comment, 121
  Use, 86
Index table
  table type, 275
  use, 276
INDEX
  Use, 209
Information message
  Use, 193
Inheritance
  ABAP Objects, 42
  Background, 166
  Reuse, 166
  Rule, 166
INITIALIZATION
  background, 304
  use, 305
INITIAL SIZE
  background, 280
  example, 281
  use, 280
INPUT
  Obsolete, 358
Input parameter
  background, 289
INSERT REPORT
  background, 325
Instance component
  Background, 162
Instance component selector
  Naming, 109, 110
Instance constructor
  Background, 171
  Use, 163
Instantiation
  ABAP Objects, 42
Integer
  background, 250
  use, 250
Interface
  ABAP Objects, 43
  Naming, 96
  Use, 167
Interface component selector
  Background, 167
  Example, 168
  Use, 167
Interface parameter
Index

K

KISS principle
  Background, 32
  Example, 33

L

Language
text environment, 333
LE
  Notation, 135
LEAVE
  Obsolete, 348
LEAVE PROGRAM
  Use, 194
LEAVE TO LIST-PROCESSING
  Background, 195
LEAVE TO TRANSACTION
  Program execution, 52
LENGTH
  Notation, 136
LIKE
  background, 230
  rule, 230
  use, 230
LIKE LINE OF
  rule, 230
LIKE reference
  Obsolete, 344
Line end comment
  Background, 115
  Use, 120
Line type
  background, 273
Line width
  Background, 89
  Example, 90
  Use, 90
List
  Background, 195, 204
  Use, 197
Literal
  background, 233
Literal operator
  Example, 91
  Use, 90
LOAD
  Obsolete, 346
  Loader
    Use, 214
LOAD-OF-PROGRAM
  background, 304
  use, 305
LOCAL
  Obsolete, 351
Local class
  Declaration, 125
  Global class, 169
  Naming, 110
  use, 221, 300
  Use, 153, 200
Local data type
  Global class, 169
Local declaration
  Arrangement, 128
  Background, 127
  Example, 129, 130
  use, 221
  Use, 169
  Validity, 129
Locale
text environment, 333
Local interface
  Declaration, 125
  Global class, 169
  Naming, 110
Localization
  Product standard, 35
Local name
  Naming, 109
Locator
  use, 316
Logical database
  Program attribute, 56
  Rule, 56
Logical Database
  Use, 57
LOOP
  Complexity, 149
LOOP AT
  background, 284
  rule, 285
LOOP AT dbtab
  Obsolete, 362
LOOP (dynpro)
  Obsolete, 358
Lowercase
Example, 88

LT
Notation, 135

M

Macro
  background, 288, 306
  example, 307, 308
  use, 307
Main program
  background, 299
  Background, 130
  Source code modularization, 131
Main program group
  background, 299
MAJOR-ID
  Obsolete, 361
Mathematical notation
  background, 247
  use, 248
MAXIMUM
  Obsolete, 359
Memory Inspector
  Memory test, 37
  use, 317
Message
  Background, 191
  Code Inspector, 73
  Exception text, 193
  Extended program check, 69
  translatability, 331
  Use, 174
MESSAGE
  background, 192
  Use, 174
Messages
  Background, 173
  Message text
    Exception text, 181
Message type
  Background, 192
Method
  background, 287
  Naming, 97
  use, 300
Method call
  Example, 142
  Long form, 142
  Notation, 141
  Short form, 141
MINIMUM
  Obsolete, 359
MINOR-ID
  Obsolete, 361
Mixed case style
  Example, 82
  Use, 81
Modern language element
  Background, 62
  Example, 64
  Use, 63
MODIFY
  example, 286
  rule, 285
MODIFY dbtab
  Obsolete, 362
Modularization
  Background, 157
MODULE
  background, 304
Module pool
  Program type, 51
  Rule, 53
  Separation of concerns, 24
  Use, 55
MOVE
  Example, 144
  Use, 143
MOVE PERCENTAGE
  Obsolete, 350
Multilingual development
  Rule, 60
MULTIPLY
  Use, 143
MULTIPLY-CORRESPONDING
  Obsolete, 352

N

Name
  Example, 100, 105
Named data object
  Background, 217
Namespace prefix
  Use, 104
Naming
  Abbreviations, 98
Index

Background, 91, 94
Digit, 99
Example, 105, 112, 113
Namespace, 108
Prefix, 111
Program object, 106
Repository object, 101
Shading, 108
Suffix, 111

Naming convention
  Background, 92
  Check, 365

Naming language
  Background, 92
  Example, 93

Native SQL
  Background, 210
  Use, 211

Nesting depth
  Background, 149
  Example, 150

NEW-SECTION
  Obsolete, 359

nmax (predefined function)
  Use, 150

nmin (predefined function)
  Use, 150

NODES
  background, 232
  interface work area, 300
  use, 233

NOT
  Notation, 135

Notation
  Example, 136, 137

Number specification
  background, 247
  example, 249

Numeric literal
  background, 234
  example, 235
  use, 234

Numeric text
  background, 243

Numeric type
  background, 249
  rule, 250

O

Object generation
  Use, 162

Object header
  dynamic memory object, 319

Object orientation
  Design, 155, 162
  Use, 157

Obsolete language element
  Background, 62, 341
  Example, 63
  Use, 63

OCCURS
  Obsolete, 345

ON CHANGE OF
  Obsolete, 350

Online Text Repository -> see OTR, 180

OO transaction
  Program execution, 52
  Use, 54

OPEN DATASET
  rule, 337

Open SQL
  Background, 210
  chained statement, 140
  Use, 211

Operator notation
  Use, 143

Original language
  Background, 60

Orthogonal concept
  ABAP Objects, 43

OTR
  Exception text, 181

Output parameter
  background, 289

P

PACK
  Obsolete, 351

Package
  Background, 101
  Naming, 95
  Naming convention, 103

Package check
  Syntax check, 67

Package concept
Index

Programming style
  Background, 79
Program object
  Naming, 106
Program structure
  Background, 79
Program type
  Background, 51
  Rule, 53
PROVIDE
  Obsolete, 356
Proxy service
  Separation of concerns, 29
Pseudo comment
  Code Inspector, 73
  Example, 75
  Extended program check, 70
  Use, 70, 73
PUBLIC SECTION
  Rule, 157

R
RAISE
  Use, 175
RAISE EXCEPTION
  Use, 176
RAISING
  MESSAGE, 192, 194
RANGES
  Obsolete, 346
READ
  Obsolete, 355
READ DATASET
  Notation, 136
READ TABLE
  background, 284
    rule, 285
READ TABLE dbtab
  Obsolete, 362
Redefinition
  Background, 166
  Use, 163
REFERENCE INTO
  dynamic access, 321
    use, 285
Reference semantics
  use, 323
Reference semantics

dynamic access, 321
REFRESH
  Obsolete, 356
REFRESH FROM dbtab
  Obsolete, 362
REJECT
  use, 303
Remote-enabled function module-> see
RFM, 54
REPLACE
  Notation, 136
  Obsolete, 353
REPLACE ... ALL OCCURENCES
  Use, 150
Report, 24
  Classical, 26
  Separation of concerns, 29
Repository object
  Naming, 101
RESUMABLE
  Use, 180
RESUME
  Use, 176
RETURN
  background, 302
    example, 303
RETURNING
  background, 289
  Naming convention, 107
    use, 290
Return value
  background, 289
RFC
  Use, 44
RFM
  Example, 24
  Use, 54
ROLLBACK WORK
  Use, 194
Rounding error
  background, 253
    example, 254
RTTC
  use, 326
RTTI
  use, 325
RTTS
  use, 325
Runtime analysis
Performance test, 37
Runtime error
  background, 257, 311
  example, 258

S

SAP buffering
  Background, 213
SAP GUI
  Background, 195
SAP List Viewer -> see ALV, 204
SAP namespace
  Name, 102
SCI -> see Code Inspector, 72
SEARCH
  Obsolete, 353
Secondary key
  example, 279
  use, 277
Security
  Code Inspector, 73
  Product standard, 36
SELECT (dynpro)
  Obsolete, 357
Selection screen
  Background, 195
  Example, 202
  Use, 197
Selection screen event
  Use, 45, 200
SELECT-OPTIONS
  Background, 195
  use, 225
Separation of concerns, 23
  Example, 28
SET EXTENDED CHECK OFF
  rule, 70
SET LOCALE
  example, 334
  text environment, 333
Shared buffer
  Use, 214
SHARED MEMORY
  Background, 213
  Use, 214
Shared Objects
  Background, 213
  Example, 215

Use, 214
Sharing
  background, 236
  details, 293
Singleton
  Background, 162
  Example, 164
  Use, 162
SLIN
  Extended program check, 69
SOA
  Separation of concerns, 24
SORT
  use, 282
SORTED BY
  background, 281
  use, 282
Sorted key
  background, 276
  use, 278
Sorted table
  background, 273
  table type, 274
  use, 275
SPLIT
  use, 336
Standalone data type
  Background, 218
  Example, 220
Standard comment
  Use, 87
Standard selection screen
  Use, 202
Standard table
  background, 273
  table type, 274
  use, 275
START-OF-SELECTION
  background, 304
  Program execution, 53
  use, 305
  Use, 45, 55
Start value
  background, 238
  example, 239
Static class
  Background, 161
  Example, 164
  Polymorphism, 163
Index

Use, 162
Static component
  Background, 162
Static constructor
  Use, 163
Status message
  Use, 193
STOP
  use, 303
Streaming
  use, 316
String
  background, 236
  dynamic data object, 313
  dynamic memory object, 315
  use, 314
String header
  dynamic memory object, 318
String literal
  background, 234
Structured programming
  Background, 157
  Use, 158
STRUCTURE typing
  Obsolete, 343, 345
Subclass
  Background, 166
  Use, 166
Subfield access
  background, 258
  use, 336
SUBMIT
  Program execution, 52
  use, 305
SUBMIT VIA JOB
  Use, 45
Subroutine
  background, 287, 288
  use, 300
  Use, 45
Subroutine pool
  Program type, 52
  Rule, 53
  Use, 54
Substructure
  background, 226
  example, 228
SUBTRACT
  Use, 143
SUBTRACT-CORRESPONDING
  Obsolete, 352
SUMMARY
  Obsolete, 358
SUMMING
  Obsolete, 359
Superclass
  Background, 166
SUPPLY
  Obsolete, 362
Surrogate area
  background, 336
sy
  background, 262
  sy-index
    use, 265
  sy-mandt
    Use, 212
Syntax check
  Background, 65
  Example, 68
  Severity, 66
  Use, 67
Syntax cleansing
  ABAP Objects, 43
Syntax error
  Background, 65
Syntax warning
  Background, 66
  Example, 68
  Priority, 66
SYST
  background, 262
System codepage
  text environment, 333
System field
  access, 262
  actual parameter, 268
  background, 262, 264
  evaluation, 265
  example, 263, 264, 266, 269, 271, 272, 273
  operand position, 272
  use, 262, 264, 265, 268, 270, 272
  user interface, 270
System text
  background, 329
  sy-subrc
    background, 267
example, 267, 268
use, 265, 267
sy-tabix
use, 265, 279

T

Table body
background, 286
dynamic memory object, 315
use, 287
Table header
dynamic memory object, 318
Table index
background, 273
TABLES
background, 232
interface work area, 300
Obsolete, 342, 344
use, 225, 233, 291
Table type
background, 273
rule, 275
Table work area
background, 232
Technical scientific notation
background, 248
Technology access
ABAP Objects, 44
Termination message
Use, 194
Terminology
Rule, 62
Text environment
background, 333
Text field
example, 237
use, 237
Text field literal
background, 234
Text file
eexample, 338
Text string
background, 236
example, 238
use, 237
Text symbol
use, 330
Time field

background, 243
Timestamp
use, 244
Top include
eexample, 225
Global declaration part, 125
Transaction
Program execution, 52
Translatability
eexample, 331, 332, 333
TRANSLATE
Obsolete, 354
Transport Organizer
ATC, 77
Truth value
background, 239
eexample, 240, 241
TRY
Background, 185
Example, 189
TYPE
background, 230
rule, 230
Type group
Background, 220
eexample, 222
Program type, 52
Rule, 53
use, 222
Use, 55
Type pool
Program type, 52
TYPE-POOLS
Obsolete, 343
TYPES
background, 220
Typing
background, 296
rule, 296

U

UI service
Separation of concerns, 29
Underscore
Example, 82
Unicode check
Example, 58
Program attribute, 56
## Index

- **Rule**, 56
- **Use**, 56
- **Unicode program**
  - **Background**, 56
  - **Example**, 59
  - **Use**, 56, 341
- **Unilingual development**
  - **Rule**, 60
- **Update function module**
  - **Use**, 54
- **Usability**
  - **Product standard**, 36
- **User interface technology**
  - **Background**, 195
  - **Encapsulation**, 199
- **USING**
  - **background**, 289
  - **use**, 290

**V**

- **Validity area**
  - **Naming convention**, 111
- **Valid value**
  - **background**, 243
  - **example**, 244
- **VALUE**
  - **DATA**, 238
- **Value range**
  - **background**, 258
- **Value semantics**
  - **dynamic access**, 321
  - **use**, 322

- **Variable**
  - **background**, 224
  - **Naming**, 96

**W**

- **Warning**
  - **Code Inspector**, 73
  - **Extended program check**, 69
  - **Use**, 193
- **Web Dynpro ABAP**
  - **Background**, 196
  - **Example**, 198
  - **Use**, 196, 207
- **WHILE**
  - **Complexity**, 149
- **WHILE VARY**
  - **Obsolete**, 350
- **WITH BYTE-ORDER MARK**
  - **OPEN DATASET**, 337
- **WRITE**
  - **Background**, 195
  - **WRITE TO**
    - **Obsolete**, 356
    - **use**, 254

**Z**

- **Zero division**
  - **background**, 255
  - **example**, 255