DEV262 – Evolution of the ABAP Programming Language

*/

Karsten Bohlmann, Holger Janz / ABAP Language Team
Thomas Jung / SAP HANA Product Management

Disclaimer

This presentation outlines our general product direction and should not be relied on in making a purchase decision. This presentation is not subject to your license agreement or any other agreement with SAP. SAP has no obligation to pursue any course of business outlined in this presentation or to develop or release any functionality mentioned in this presentation. This presentation and SAP's strategy and possible future developments are subject to change and may be changed by SAP at any time for any reason without notice. This document is provided without a warranty of any kind, either express or implied, including but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. SAP assumes no responsibility for errors or omissions in this document, except if such damages were caused by SAP intentionally or grossly negligent.

Agenda

Inline Declarations

Constructor Expressions

Table Expressions: Selection, Table Comprehension, and Reduction

Grouping

Further News

Simplification is the Goal

7.40 ABAP extensions address “programming **in the small**”.

- Not prescribing some new “paradigm” or “design methodology”.
- But you will find them useful “every 2nd line“.
- You can write ABAP code more similar to state-of-the-art languages.

Simplification is the overall goal. Features allow you to:

- **Avoid redundancies**
  - Inline declarations
  - Type inference
- Express pure **value** computations concisely **w/o state** manipulation
  - Constructor expressions
  - Table comprehensions
- Leverage **powerful operations** on data collections
  - Grouping

Previously on ABAP News

Expressions (ABAP 7.02 and up)

```
o->f( p1 = i + 1  p2 = g( ) ).
```

Computation in Operands

```
IF o->m1( )->intf~m2( )->a > 1.
```

Method Call Chains

```
s = to_upper( condense( str ) ) &&
    concat_lines_of( strtab ).
```

Expressions for String Operations

```
s = |Hello,\n| &
    |today is { sy-datum date = iso }\n| &
    |now is { sy-uzeit time = iso }|.
```

String Templates

Inline Declarations

Expressions
Inline Declarations

Inline Declaration

```
... DATA(var) ...
... FIELD-SYMBOL(<fs>) ...
```

These new operators combine the declaration and (initial) value assignment of a variable / field symbol.

Allowed at write positions for which a type can be determined statically from the context.
This “inferred” type is given to the declared symbol.

Expressions

Inline Declarations – DATA( )

```
LOOP AT tab INTO DATA(line).
  …
ENDLOOP.
```

```
READ TABLE tab … INTO DATA(line).
```

```
oref->meth( IMPORTING
              p1 = DATA(a1)
              p2 = DATA(a2)
            RECEIVING
              rc = DATA(ok) ).
```

```
DATA(new) = old + inc.
```

```
FIND … IN … MATCH COUNT  DATA(cnt)
            MATCH OFFSET DATA(off).
```

```
CALL TRANSFORMATION … RESULT XML DATA(xml).
```

Expressions

Inline Declarations – DATA( )

```
DATA ixml        TYPE REF TO if_ixml.
DATA document    TYPE REF TO if_ixml_document.
DATA streamfact TYPE REF TO if_ixml_stream_factory.
DATA istream     TYPE REF TO if_ixml_istream.

ixml         = cl_ixml=>create( ).
document     = ixml->create_document( ).

streamfact   = ixml->create_stream_factory( ).
stream       = streamfact->create_istream_xstring( … ).
```

```
DATA(ixml)       = cl_ixml=>create( ).
DATA(document)   = ixml->create_document( ).

DATA(stream)     = ixml->create_stream_factory(
                       )->create_istream_xstring( … ).
```

Expressions

Inline Declarations – FIELD-SYMBOL( )

```
FIELD-SYMBOLS: <line1> LIKE LINE OF tab,
               <line>  LIKE LINE OF tab,
               <comp>  TYPE t.

READ TABLE tab … ASSIGNING <line1>.

LOOP AT tab ASSIGNING <line>.

  ASSIGN COMPONENT … OF <line> TO <comp>.
  …
ENDLOOP.
```

```
READ TABLE tab … ASSIGNING FIELD-SYMBOL(<line1>).

LOOP AT tab ASSIGNING FIELD-SYMBOL(<line>).

  ASSIGN COMPONENT … OF <line> TO FIELD-SYMBOL(<comp>).
  …
ENDLOOP.
```

Constructor Expressions

Expressions
Constructor Expressions

Syntax: `operator type( … )`
`operator #( … )`

These new operators construct values of a specified *type* or inferred type (#), where *operator* is one of { **NEW**, **VALUE**, … } and the syntax inside `( )` depends on *operator.*

- **NEW** creates objects / data objects
- **VALUE** creates values (esp. of structured types)
- **REF** creates data references
- **CONV** converts values
- **CAST** performs up or down casts of references
- **EXACT** performs lossless calculations or assignments
- **COND** and **SWITCH** compute values conditionally
- **CORRESPONDING** maps/moves components in structured types

Expressions

Constructor Expressions – Operator NEW( )

```
DATA dref TYPE REF TO data.
FIELD-SYMBOLS <fs> TYPE t_struc.
CREATE DATA dref TYPE t_struc.
ASSIGN dref->* TO <fs>.
<fs>-c1 = 10.
<fs>-c2 = 'a'.
```

```
DATA dref TYPE REF TO data.
dref = NEW t_struc( c1 = 10 c2 = 'a' ).
```

```
DATA oref TYPE REF TO class.
CREATE OBJECT oref
  EXPORTING p1 = a1 p2 = a2.
```

```
DATA(oref) = NEW class( p1 = a1 p2 = a2 ).
```

Expressions

Constructor Expressions – Operator VALUE( )

```
DATA tab TYPE t_tab.
DATA line LIKE LINE OF tab.
line-col1 = 10.
line-col2 = 'a'.
APPEND line TO tab.
line-col2 = 'b'.
APPEND line TO tab.
APPEND LINES OF othertab TO tab.

method( tab ).
```

```
DATA(tab) = VALUE t_tab(
  ( col1 = 10 col2 = 'a' )
  ( col1 = 10 col2 = 'b' )
  ( LINES OF othertab ) ).
method( itab ).
```

```
method( VALUE #(
  col1 = 10
   ( col2 = 'a' )
   ( col2 = 'b' )
  ( LINES OF othertab ) ) ).
```

Expressions

Constructor Expressions – Operator REF( )

```
DATA dref TYPE REF TO t_param.
GET REFERENCE OF param INTO dref.

DATA(ptab) = VALUE abap_parmbind_tab(
  ( name  = name
    kind  = cl_abap_objectdescr=>exporting
    value = dref ) ).

CALL METHOD (class)=>(meth) PARAMETER-TABLE ptab.
```

```
DATA(ptab) = VALUE abap_parmbind_tab(
  ( name  = name
    kind  = cl_abap_objectdescr=>exporting
    value = REF #( param ) ) ).

CALL METHOD (class)=>(meth) PARAMETER-TABLE ptab.
```

Expressions

Constructor Expressions – Operator CONV( )

```
DATA helper TYPE string.
helper = sy-uname.
DATA(xstr) = cl_abap_codepage=>convert_to( source = helper ).
```

```
DATA(xstr) = cl_abap_codepage=>convert_to( source = CONV #( sy-uname ) ).
```

```
DATA(i) = 1 / 4.
```

```
DATA(df) = CONV decfloat34( 1 / 4 ).
```

```
CHECK ` ` = abap_false.
```

```
CHECK CONV abap_bool( ` ` ) = abap_false.
```

Expressions

Constructor Expressions – Local Variable Bindings

Several constructor operators allow the binding of intermediate results to expression-local variables.

```
DATA(sqsize) = CONV i( LET s = get_size( … )
                       IN  s * s ).
```

```
oref = NEW class( LET s0 = get_size( … )
                      s  = COND #( WHEN size_ok( s0 ) THEN s0 ELSE 1 )
                  IN  p_size_x = s
                      p_size_y = s ).
```

Expressions

Constructor Expressions – Operator CAST( )

```
DATA structdescr TYPE REF TO cl_abap_structdescr.
structdescr ?= cl_abap_typedescr=>describe_by_name( 'T100' ).
DATA(components) = structdescr->components.
```

```
DATA(components) = CAST cl_abap_structdescr(
  cl_abap_typedescr=>describe_by_name( 'T100' )
 )->components.
```

Expressions

Constructor Expressions – Operator EXACT( )

```
TYPES numtext TYPE n LENGTH 255.
DATA number TYPE numtext.
TRY.
    MOVE EXACT '4 Apples + 3 Oranges' TO number.
  CATCH cx_sy_conversion_error INTO DATA(exc).
    ...
ENDTRY.
```

```
TYPES numtext TYPE n LENGTH 255.
TRY.
    DATA(number) = EXACT numtext( '4 Apples + 3 Oranges' ).
  CATCH cx_sy_conversion_error INTO DATA(exc).
    ...
ENDTRY.
```

Expressions

Constructor Expressions – Operator COND( )

```
DATA time TYPE string.
DATA(now) = sy-timlo.
IF now < '120000'.
  time =  |{ now TIME = ISO } AM|.
ELSEIF now > '120000'.
  time = |{ CONV t( now - 12 * 3600 )
            TIME = ISO } PM|.
ELSEIF now = '120000'.
  time = `High Noon`.
ELSE.
  RAISE EXCEPTION TYPE cx_cant_be.
ENDIF.
```

```
DATA(time) =
  COND string(
    LET  now = sy-timlo IN
    WHEN now < '120000' THEN
      |{ now TIME = ISO } AM|
    WHEN now > '120000' THEN
      |{ CONV t( now - 12 * 3600 )
         TIME = ISO } PM|
    WHEN now = '120000' THEN
      `High Noon`
    ELSE
      THROW cx_cant_be( ) ).
```

Expressions

Constructor Expressions – Operator SWITCH( )

```
DATA number TYPE string.
CASE sy-index.
  WHEN 1.
    number = 'one'.
  WHEN 2.
    number = 'two'.
  WHEN 3.
    number = 'three'.
  WHEN OTHERS.
    RAISE EXCEPTION TYPE cx_overflow.
ENDCASE.
```

```
DATA(number) =
  SWITCH string( sy-index
                 WHEN 1 THEN 'one'
                 WHEN 2 THEN 'two'
                 WHEN 3 THEN 'three'
                 ELSE THROW cx_overflow( ) ).
```

Expressions

Constructor Expressions – Operator CORRESPONDING( )

This operator applies name-matching recursively to structures and tables.

Components with distinct names can be mapped explicitly.

Components with equal names can be excluded from mapping.

```
LOOP AT itab1 ASSIGNING <line1>.
  MOVE-CORRESPONDING <line1> TO line2.
  line2-foo = <line1>-bar.
  CLEAR line2-baz.
  INSERT line2 INTO TABLE itab2.
ENDLOOP.
```

```
itab2 = CORRESPONDING #( itab1
                         MAPPING foo = bar
                         EXCEPT  baz ).
```

Exercise

#SAPdev

Table Expressions

Expressions
Table Expressions

Syntax: `tab[ … ]`

- Read access to internal tables at all expression-enabled operand positions.
- Also enabled in write positions.

Expressions

Table Selection – Specification of Table Line

```
READ TABLE tab INDEX idx INTO wa.
```

```
wa = tab[ idx ].
```

```
READ TABLE tab INDEX idx
  USING KEY key INTO wa.
```

```
wa = tab[ KEY key INDEX idx ].
```

```
READ TABLE tab WITH KEY
  col1 = … col2 = … INTO wa.
```

```
wa = tab[ col1 = … col2 = … ].
```

```
READ TABLE tab WITH TABLE KEY key
  COMPONENTS col1 = … col2 = …
  INTO wa.
```

```
wa = tab[ KEY key col1 = … col2 = … ].
```

Expressions

Table Selection – Retrieval Method

```
READ TABLE tab … ASSIGNING <line>.
<line>-col = 10.
```

```
tab[ … ]-col = 10.
```

```
READ TABLE tab … INTO line.
IF sy-subrc <> 0.
  line = line0.
ENDIF.
meth( line ).
```

```
meth( VALUE #( tab[ … ] DEFAULT line0 ) ).
```

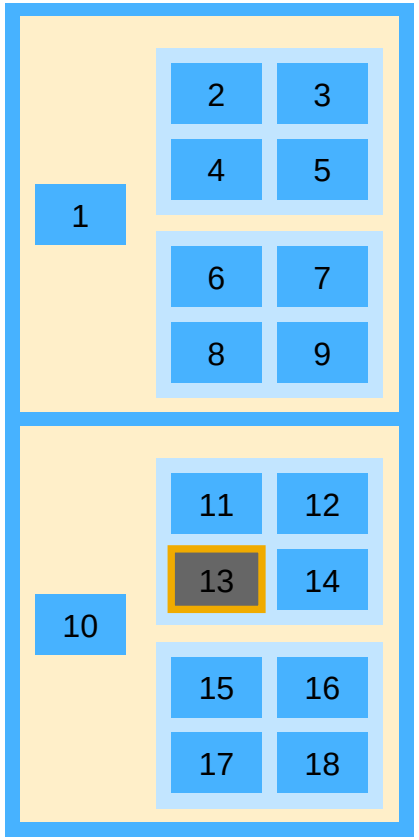
SP8

```
READ TABLE tab … REFERENCE INTO lineref.
meth( lineref ).
```

```
meth( REF #( tab[ … ] ) ).
```

Expressions

Table Selection – Chaining




```
READ TABLE tab      INTO DATA(wa1) INDEX 2.
READ TABLE wa1-col2 INTO DATA(wa2) INDEX 1.
READ TABLE wa2      INTO DATA(wa3) INDEX 2.
DATA(num) = wa3-col1.
```

```
DATA(num) = tab[ 2 ]-col2[ 1 ][ 2 ]-col1.
```

Expressions

Table Selection – Built-in Functions

```
READ TABLE tab … TRANSPORTING NO FIELDS.
DATA(idx) = sy-tabix.
```

```
DATA(idx) = line_index( tab[ … ] ).
```

```
READ TABLE tab … TRANSPORTING NO FIELDS.
IF sy-subrc = 0.
  …
ENDIF.
```

```
IF line_exists( tab[ … ] ).
 …
ENDIF.
```

Expressions

Table Comprehensions – Building Tables Functionally

Table-driven:

```
VALUE tabletype( FOR line IN tab WHERE ( … )
                   ( … line-… … line-… … )
                 )
```

For each selected line in the source table(s), construct a line in the result table.

Generalization of value constructor from static to dynamic number of lines

Condition-driven:

```
VALUE tabletype( FOR i = 1                  WHILE i < n
                   FOR j = 10 THEN j - 1 UNTIL j = 0
                     ( … i … j … )
                 )
```

SP8

Construct lines in the result table until the condition on the loop variable is false.

Expressions
Table Comprehensions - Example

Build “cross product” of two tables.

```
LOOP AT tab1 INTO line1 WHERE val = 1.
  LOOP AT tab2 ASSIGNING <line2>.
    line3-c = f( line1-a ).
    line3-d = <line2>-b.
    APPEND line3 TO tab1_x_tab2.
  ENDLOOP.
ENDLOOP.
```

```
tab1_x_tab2 = VALUE #(
  FOR line1   IN tab1 WHERE ( val = 1 )
    FOR <line2> IN tab2
      ( c = f( line1-a ) d = <line2>-b ) ).
```

Expressions

Reductions – Computing Aggregate Values Functionally

Syntax:

```
REDUCE type( INIT res = …
             hlp = …
           FOR  var1 …
           NEXT hlp = … hlp … var1 … res … )
```

In each iteration (table-driven or condition-driven), re-compute the result value (and helper variables).

Generalization of table comprehension from table to arbitrary result type

Expressions
Reductions - Example

Render columns of an internal table into an HTML string.

```
DATA htmltab TYPE string.
htmltab = `<table>`.
LOOP AT tab INTO line.
  htmltab =
   |{ htmltab }<tr><td>{ line-a }| &
   |</td><td>{ line-b }</td></tr>|.
ENDLOOP.
htmltab = htmltab && `</table>`.
```

```
DATA(htmltab) = REDUCE string(
 INIT h = `<table>`
 FOR  line IN tab
 NEXT h = |{ h }<tr><td>{ line-a }</td>| &
          |<td>{ line-b }</td></tr>|
 ) && `</table>`.
```

Grouping

Grouping

Processing Table Lines in Groups – Statement Variant

```
LOOP AT tab bind(x)
  WHERE …
  GROUP BY groupkey(x)
  bind(g).
    …
    LOOP AT GROUP g bind(m).
      …
    ENDLOOP.
  …
ENDLOOP.
```

***bind*(x)** is a binding (**INTO** / **ASSIGNING**) of a variable **x** for the original lines in the table being grouped

***bind*(g)** is a binding of a variable **g** for the group key values

***groupkey*(x)** is either
– an expression over **x**
– or a tuple **(*name*$_1$ = *expr*$_1$ *name*$_2$ = *expr*$_2$ …)** over **x**

The **GROUP BY** loop iterates through
– the computed group key values in **g**, if ***bind*(g)** is given
– representative group members in **x**, if ***bind*(g)** is missing

The **LOOP AT GROUP** iterates through all members of the current group **g** (in **m**).

Grouping

Processing Table Lines in Groups – Expression Variant

Table Comprehension

```
VALUE #( FOR GROUPS g OF x IN tab GROUP BY groupkey(x)
           ( … FOR m IN GROUP g ( … ) … )
         )
```

Reduction

```
REDUCE #( INIT r = …
            FOR GROUPS g OF x IN tab GROUP BY groupkey(x)
            NEXT r = … FOR m IN GROUP g ( … ) …
          )
```

Grouping

Processing Table Lines in Groups - Example

Group flights by duration.

```
LOOP AT flights ASSIGNING FIELD-SYMBOL(<flg>)
 GROUP BY COND #( WHEN <flg>-fltime < 120 THEN  0
                  WHEN <flg>-fltime > 600 THEN 99
                  ELSE trunc( <flg>-fltime / '60' ) )
 ASCENDING
 ASSIGNING FIELD-SYMBOL(<fd>).
  WRITE / |Duration: { COND #( WHEN <fd> =  0 THEN `less than 2`
                               WHEN <fd> = 99 THEN `more than 10`
                               ELSE <fd> ) } hours|.
  LOOP AT GROUP <fd> ASSIGNING <flg>.
    WRITE: |{ <flg>-cityfrom }-{ <flg>-cityto }: { <flg>-fltime }|.
  ENDLOOP.
ENDLOOP.
```

Simplified ABAP Code Through Expressions

ABAP 7.31

```
DATA: it1 TYPE tt1,
      wa1 LIKE LINE OF it1.

wa1-a = 7. wa1-b = 9.
INSERT wa1 INTO TABLE it1.
wa1-a = 3. wa1-b = 5.
INSERT wa1 INTO TABLE it1.

FIELD-SYMBOLS <wa1> LIKE LINE OF it1.
DATA wa2 LIKE LINE OF it2.
LOOP AT it1 ASSIGNING <wa1>.
  wa2 = t1_to_t2( <wa1> ).
  INSERT wa2 INTO TABLE it2.
ENDLOOP.

DATA: ref TYPE REF TO class1,
      tmp TYPE string.
READ TABLE it1 WITH KEY a = 3 ASSIGNING <wa1>.
CREATE OBJECT ref.
tmp = <wa1>-b.
ref->do_something( tmp ).
```

ABAP 7.40

```
" no redundant variable declarations

" value constructor expression (for table);
" inline declaration with inferred type
DATA(it1) = VALUE tt1( ( a = 7 b = 9 )
                       ( a = 3 b = 5 ) ).

" table comprehension
it2 = VALUE #( FOR <wa1> IN it1
               ( t1_to_t2( <wa1> ) ) ).


" object creation; table selection;
" value conversion to inferred type
NEW class1( )->do_something(
  CONV #( it1[ a = 3 ]-b ) ).
```

Expressions

ABAP is Extensively Expression Enabled – Pitfalls

Obfuscation

```
lcl=>cm_ii(
  lcl=>factory2(
    f_in1 = lcl=>factory0( )->im_ii( i )
    f_in2 = lcl=>cm_ii( 2 ** i ) - 3 +
            itab[ 2 ]-s-xtab[ x = 'abc' ]
  )->m_ii(
       lcl=>factory1( f_in = itab[ a = 2
                                   b = 4711 / 2
                                 ]-x
       )->m_lif_i( lcl=>cm_ii( 5 ) )->im_ii( 6 )
  )
).
```

Performance

```
itab1[ 1 ]-a = itab2[ 1 ]-x.
itab1[ 1 ]-b = itab2[ 1 ]-y.
itab1[ 1 ]-c = itab2[ 1 ]-z.
```

Exercise

#SAPdev

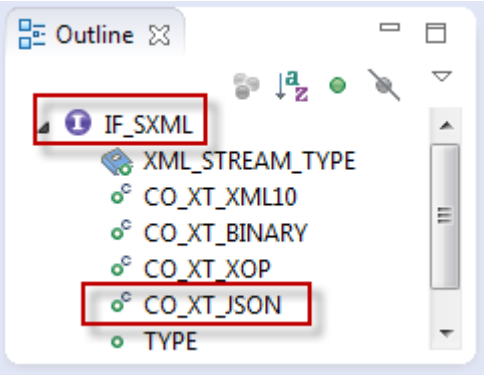
Further News

#SAPdev

JSON

ABAP and JSON

JSON Support in sXML-Library



Backported to 7.02!

JSON - JavaScript Object Notation, data format in text form for data exchange.

JSON-XML - SAP-specific representation of JSON data in XML format

asJSON - ABAP Serialization JSON,
Canonical JSON representation for serialization/deserialization of ABAP data by transformation ID

JSON

Readers and Writers

JSON to JSON-XML and Vice Versa

```
DATA(json) = cl_abap_codepage=>convert_to( `{"TEXT":"JSON"}` ).
DATA(json_reader) = cl_sxml_string_reader=>create( json ).
DATA(xml_writer) = cl_sxml_string_writer=>create( ).
json_reader->next_node( ).
json_reader->skip_node( xml_writer ).
cl_demo_output=>display_xml( xml_writer->get_output( ) ).
```

Output

```
<object>
 <str name="TEXT">JSON</str>
</object>
```

```
DATA(xml) = cl_abap_codepage=>convert_to(
  `<object><str name="TEXT">JSON</str></object>` ).
DATA(xml_reader) = cl_sxml_string_reader=>create( xml ).
DATA(json_writer) = cl_sxml_string_writer=>create( type = if_sxml=>co_xt_json ).
xml_reader->next_node( ).
xml_reader->skip_node( json_writer ).
cl_demo_output=>display_json( json_writer->get_output( ) ).
```

Output

```
{
 "TEXT":"JSON"
}
```

JSON

Transformation ID

JSON to JSON-XML and Vice Versa

```
DATA(json) = `{"TEXT":"JSON"}`.
CALL TRANSFORMATION id SOURCE XML json
                       RESULT XML DATA(xml).
cl_demo_output=>display_xml( xml ).
```

Output

```
<object>
 <str name="TEXT">JSON</str>
</object>
```

```
DATA(xml) = `<object><str name="TEXT">JSON</str></object>`.
DATA(json_writer) = cl_sxml_string_writer=>create( type = if_sxml=>co_xt_json ).
CALL TRANSFORMATION id SOURCE XML xml
                       RESULT XML json_writer.
cl_demo_output=>display_json( json_writer->get_output( )  ).
```

Output

```
{
 "TEXT":"JSON"
}
```

JSON

asJSON

ABAP to JSON and Vice Versa

```
DATA(text) = `JSON`.
DATA(json_writer) = cl_sxml_string_writer=>create( type = if_sxml=>co_xt_json ).
CALL TRANSFORMATION id SOURCE text = text
                       RESULT XML json_writer.
cl_demo_output=>display_json( json_writer->get_output( ) ).
```

Output

```
{
 "TEXT":"JSON"
}
```

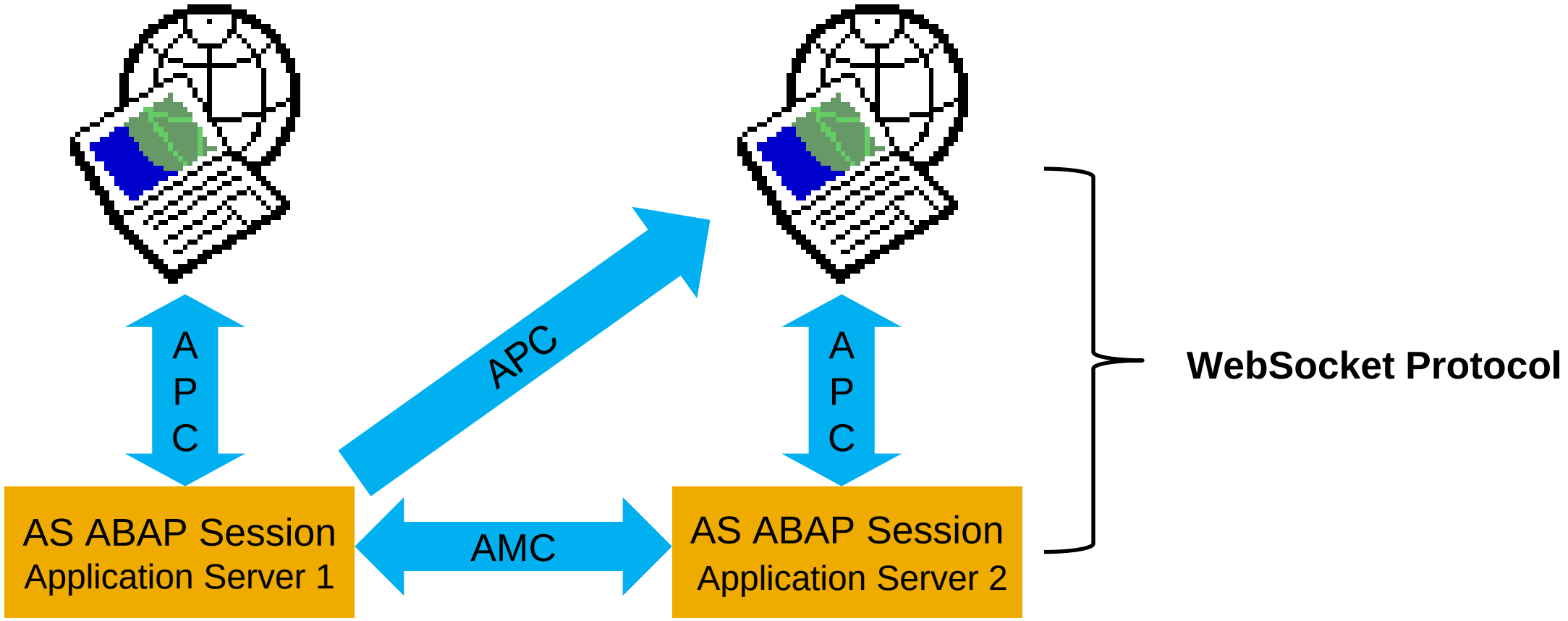
```
DATA(json) = `{"TEXT":"JSON"}`.
DATA text TYPE string.
CALL TRANSFORMATION id SOURCE XML json
                       RESULT text = text.
cl_demo_output=>display( text ).
```

Output

```
JSON
```

ABAP Channels

ABAP Messaging Channels (AMC) and ABAP Push Channels (APC)



ABAP Channels
Characteristcs

ABAP Push Channel (APC)
- Integration of WebSocket (RFC 6455) into ABAP
- **Outlook**: SAPGUI Push Channel

ABAP Messaging Channel (AMC)
- Inter-Session Messaging (Pub/Sub) in ABAP
- Binding AMC channels to ABAP sessions

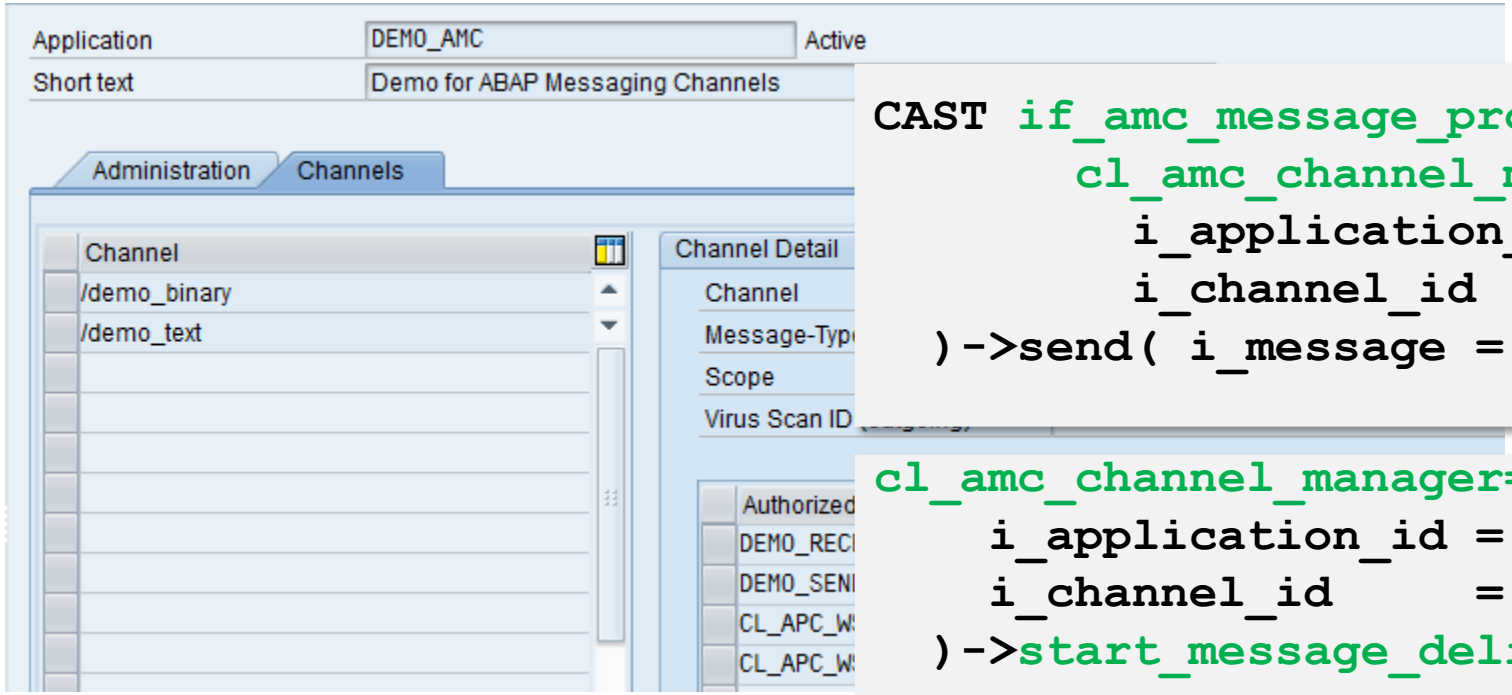
Collaboration (AMC & APC)
- Binding AMC channels to APC (WebSocket) Connections

Goal: Replace polling by event-driven approach whenever possible

ABAP Channels

ABAP Messaging Channels

Enable message based communications between ABAP programs across the boundaries of application servers.




```
CAST if_amc_message_producer_text(
       cl_amc_channel_manager=>create_message_producer(
         i_application_id = 'DEMO_AMC'
         i_channel_id     = '/demo_text' )
  )->send( i_message = ... ).
```

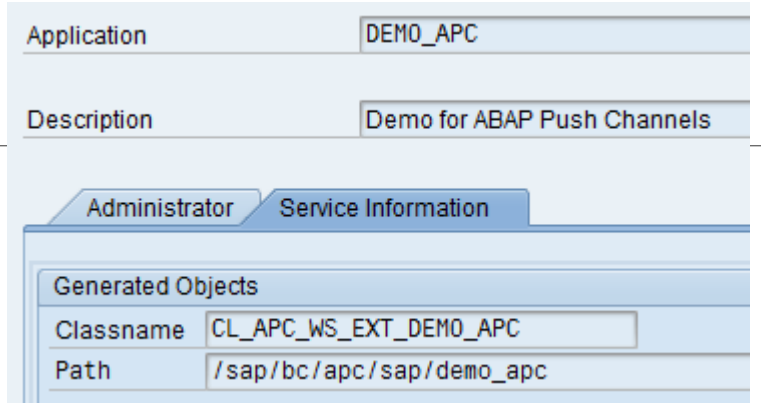
```
cl_amc_channel_manager=>create_message_consumer(
    i_application_id = 'DEMO_AMC'
    i_channel_id     = '/demo_text'
  )->start_message_delivery( i_receiver = receiver ).

WAIT FOR MESSAGING CHANNELS
     UNTIL receiver->text_message IS NOT INITIAL
     UP TO 60 SECONDS.
```

ABAP Channels

ABAP Push Channels – ABAP side

Enable bidirectional communications between ABAP programs and the internet using the WebSocket protocol. APCs can be connected to AMCs.




```
METHOD if_apc_ws_extension~on_start.
  ...
  IF amc_flag = abap_true.
    TRY.
        i_context->get_binding_manager(
          )->bind_amc_message_consumer(
            i_application_id =  'DEMO_AMC'
            i_channel_id     = '/demo_text' ).
      CATCH cx_apc_error INTO DATA(exc).
        MESSAGE exc->get_text( ) TYPE 'X'.
    ENDTRY.
  ELSE.
    "Default behavior
  ENDIF.
ENDMETHOD.
```

```
METHOD if_apc_ws_extension~on_message.
...
IF amc_flag = abap_true.
  CAST if_amc_message_producer_text(
        cl_amc_channel_manager=>create_message_producer(
          i_application_id = 'DEMO_AMC'
          i_channel_id     = '/demo_text' )
           )->send( i_message = ...  ).
ELSE.
  DATA(message_manager) =
    i_message->get_context( )->get_message_manager( ).
  DATA(message) = message_manager->create_message( ).
  message->set_text( ... ).
  message_manager->send( message ).
ENDIF.
```

ABAP Channels

ABAP Push Channels – Internet side

```
if (para == "open" )
{
   ws = new WebSocket("ws://ux4317.wdf.sap.corp:50018/sap/bc/apc/sap/demo_apc?amc=x");
};
if (para == "send" )
{
   ws.send( ">" + document.getElementById("input").value + "<" +
            " from " + window.location.host + window.location.pathname );
};
if (para == "close" )
{
   ws.close( );
};
ws.onopen = function()
{
   alert("WebSocket opened");
};
ws.onmessage = function (evt)
{
   var received_msg = evt.data;
   alert(evt.data);
};
ws.onclose = function()
{
   alert("WebSocket closed");
};
```

WebSocket Communication with ABAP Push Channel

Open WebSocket

Send message to APC Hello APC!

Close WebSocket

Method IF_APC_WS_EXTENSION~ON_START

```
13
14    IF amc_flag = abap_true.
15      TRY.
16          i_context->get_binding_manager(
17            )->bind_amc_message_consumer(
```

Method IF_APC_WS_EXTENSION~ON_MESSAGE Active

```
17          IF amc_flag = abap_true.
18            CAST if_amc_message_producer_text(
19                  cl_amc_channel_manager=>create_message_producer(
20                    i_application_id = 'DEMO_AMC'
21                    i_channel_id     = '/demo_text' )
22             )->send( i_message = `AMC-` && msg  ) ##NO_TEXT.
```

Further News
Security Checks

“SLIN_SEC”

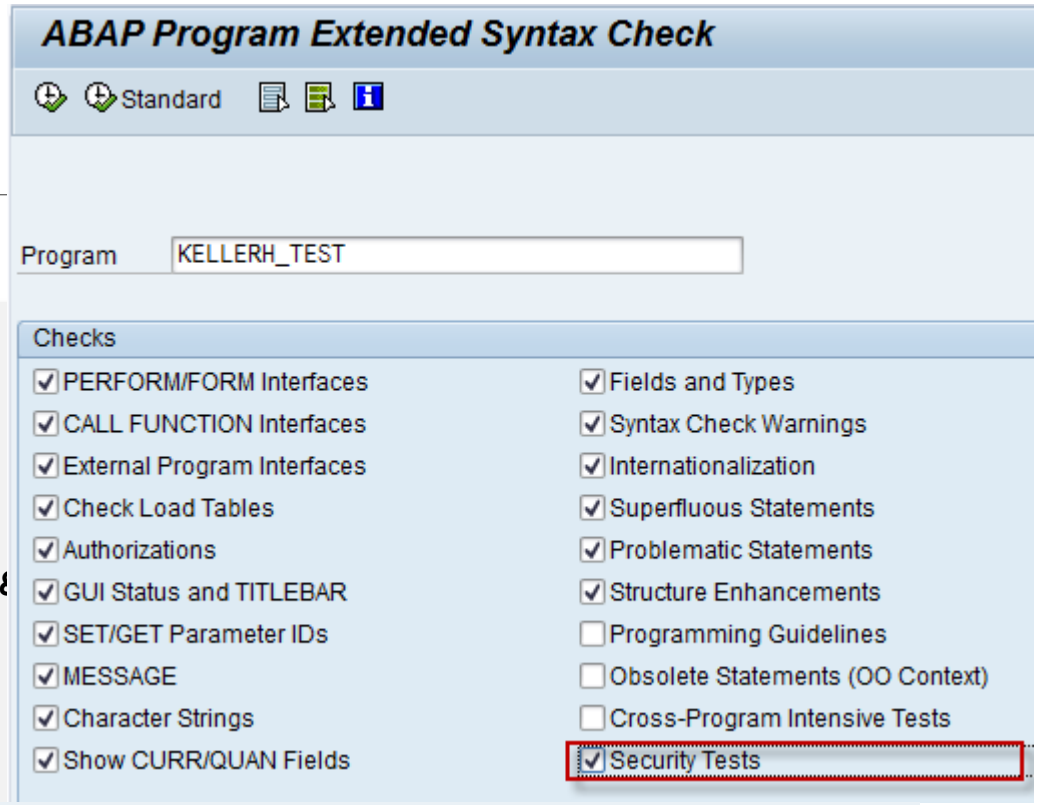
```
DATA name TYPE string.
DATA customers TYPE TABLE OF scustom WITH EMPTY KEY.

cl_demo_input=>request( CHANGING field = name ).

DATA(cond) = `country = 'DE' AND name = '` && name &&

TRY.
    SELECT * FROM scustom
             INTO TABLE customers
             WHERE (cond).
    cl_demo_output=>display( customers ).
  CATCH cx_sy_dynamic_osql_syntax.
    cl_demo_output=>display( 'Wrong input' ).
ENDTRY.
```

```
DATA(cond) = `country = 'DE' AND name = '` &&
             cl_abap_dyn_prg=>escape_quotes( name ) && `'`.
```



Check for Program KELLERH_TEST	Error	Warnings	Messages
Test Environment	0	0	0
Security Tests	1	0	0
Hidden Errors an			

Security Tests

* * * * * * * * * * * * * * * * * * * E R R O R S * * * * * * * * * * *

Program: KELLERH_TEST Row: 12 [Prio 1]
Parameter COND for statement SELECT allows an SQL injection.
Data Flow:
Procedure Call: REQUEST Parameter: NAME (REPS KELLERH_TEST [6])
NAME -> COND (include KELLERH_TEST, line 8)
Cannot be hidden using a pragma.

Further News

ABAP Doc for ABAP in Eclipse

Inline Documentation of Source Code based Development Objects

```
5  "! Demo usage of ABAP Doc
6  CLASS demo DEFINITION.
7    PUBLIC SECTION.
8      "! Divides p1 by p2 and returns the result
9      "! rounded to <em>2 decimal places</em> after the decimal separator.
10     "! @parameter p1 | Number
11     "! @parameter p2 | Quotient
12     "! @parameter r  | Result
13     "! @raising   cx_sy_arithmetic_error | Error in calculation
14     METHODS meth
15       IMPORTING
16        p1 TYPE i
17        p2 TYPE i
18       RETURNING
19         VALUE(r) TYPE decfloat34
20       RAISING
21         cx_sy_arithmetic_error.
22 ENDCLASS.
23
24 CLASS demo IMPLEMENTATION.
25   METHOD meth.
26     "! Decimal places.
27     CONSTANTS decimals TYPE i VALUE i`2`. r = p1 / p2.
28     r = round( val = r dec = decimals ).
29   ENDMETHOD.
30 ENDCLASS.
31
32 DATA oref TYPE REF TO demo.
33
34 START-OF-SELECTION.
35
36   oref->meth( p1 = 2 p2 = 3 ).
```

F2

demo
meth

Documentation
Demo usage of ABAP Doc

Components
meth Divides p1 by p2 and returns the result rounded to *2 decimal places* after the decimal separator.

meth
importing **p1** type i
p2 type i
returning value(**r**) type decfloat34
raising **cx_sy_arithmetic_error**

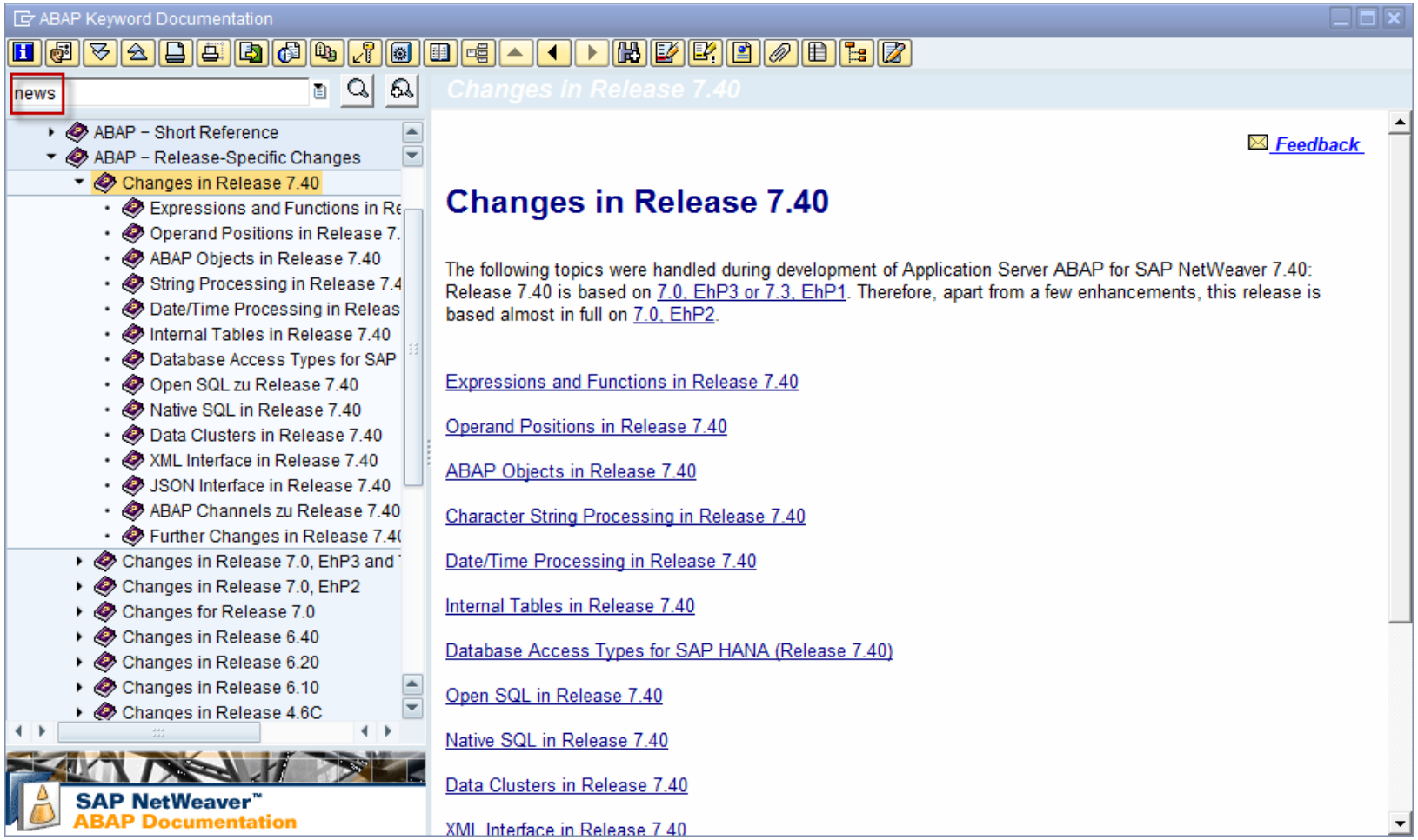
Documentation
Divides p1 by p2 and returns the result rounded to *2 decimal places* after the decimal separator.

Parameters
p1 Number
p2 Quotient
r Result

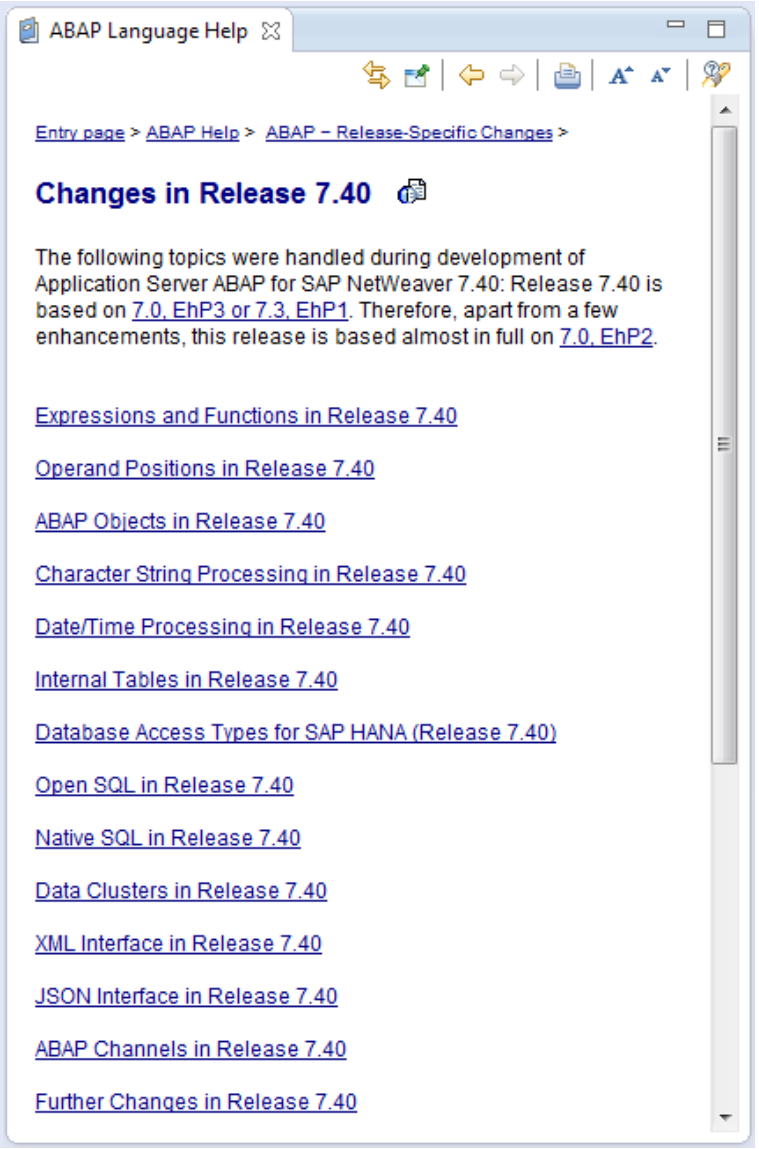
Class-based exceptions
cx_sy_arithmetic_error Error in calculation

Documentation

See ABAP Keyword Documentation



Classic



ADT

Further Information

SAP Public Web

scn.sap.com/community/abap/blog/2013/07/22/abap-news-for-release-740 - ABAP 7.4 News Blog
scn.sap.com/community/abap - ABAP Community
help.sap.com/abapdocu_740/en - ABAP Online Documentation/Reference 7.4
scn.sap.com/docs/DOC-41566 - SCN Trail Editions: SAP NetWeaver Application Server 7.4

SAP Education and Certification Opportunities

www.sap.com/education

Watch SAP d-code Online

www.sapcode.com/online

Thank you

Contact information:

Thomas Jung / thomas.jung@sap.com

#SAPdev