eBook Gratuit

APPRENEZ ABAP

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#abap

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: abap

It is an unofficial and free ABAP ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official ABAP.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec ABAP

Remarques

ABAP est un langage de programmation développé par SAP pour la programmation d'applications métier dans l'environnement SAP.

Auparavant uniquement procédural, ABAP est maintenant aussi un langage orienté objet grâce à l'amélioration des objets ABAP.

Versions

Version	Date de sortie
ABAP 7.50	2015-10-20
ABAP 7.40	2012-11-29
ABAP 7.0	2006-04-01
ABAP 6.40	2004-04-01
ABAP 6.20	2002-04-01
ABAP 6.10	2001-07-01
ABAP 4.6C	2001-04-01
ABAP 4.6A	1999-12-01
ABAP 4.5	1999-03-01
ABAP 4.0	1998-06-01
ABAP 3.0	1997-02-20

Examples

Bonjour le monde

```
PROGRAM zhello_world.
START-OF-SELECTION.
    WRITE 'Hello, World!'.
```

Au lieu d'imprimer sur la console, ABAP écrit les valeurs dans une liste qui sera affichée dès que la logique principale aura été exécutée.

Bonjour tout le monde dans les objets ABAP

```
PROGRAM zhello_world.

CLASS main DEFINITION FINAL CREATE PRIVATE.
  PUBLIC SECTION.
    CLASS-METHODS: start.
ENDCLASS.

CLASS main IMPLEMENTATION.
  METHOD start.
    cl_demo_output=>display( 'Hello World!' ).
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  main=>start( ).
```

Lire Démarrer avec ABAP en ligne: https://riptutorial.com/fr/abap/topic/1196/demarrer-avec-abap

Chapitre 2: ABAP GRID List Viewer (ALV)

Examples

Créer et afficher une ALV

Cet exemple dépeint la création ALV la plus simple à l'aide de la classe `cl_salv_table` et aucune option de formatage supplémentaire. Des options de formatage supplémentaires seraient incluses après le bloc `TRY ENDTRY` et avant l' `alv->display( )` la `alv->display( )` .

Tous les exemples suivants utilisant l'approche Objets ABAP pour la création d'ALV utiliseront cet exemple comme point de départ.

```
DATA: t_spfli       TYPE STANDARD TABLE OF spfli,
      alv           TYPE REF TO cl_salv_table,
      error_message TYPE REF TO cx_salv_msg.

" Fill the internal table with example data
SELECT * FROM spfli INTO TABLE t_spfli.

" Fill ALV object with data from the internal table
TRY.
    cl_salv_table=>factory(
      IMPORTING
        r_salv_table = alv
      CHANGING
        t_table      = t_spfli ).
  CATCH cx_salv_msg INTO error_message.
    " error handling
ENDTRY.

" Use the ALV object's display method to show the ALV on the screen
alv->display( ).
```

Optimiser la largeur de colonne ALV

Cet exemple montre comment optimiser la largeur de la colonne afin que les en-têtes de colonne et les données ne soient pas tronqués.

```
alv->get_columns( )->set_optimize( ).
```

Masquer les colonnes dans un ALV

Cet exemple masque le `MANDT` (client) de l'ALV. Notez que le paramètre passé à `get_column( )` *doit* être mis en majuscule pour que cela fonctionne.

```
alv->get_columns( )->get_column( 'MANDT' )->set_visible( if_salv_c_bool_sap=>false ).
```

Renommer les en-têtes de colonne dans une ALV

Le texte de la colonne peut changer lors du redimensionnement horizontal d'une colonne. Il existe trois méthodes pour y parvenir:

Nom de la méthode	Longueur maximale du cap
`set_short_text`	dix
`set_medium_text`	20
`set_long_text`	40

L'exemple suivant montre l'utilisation des trois. Un objet `column` est déclaré et instancié comme référence au résultat de `alv->get_columns( )->get_column( 'DISTID' )` . Le nom de la colonne *doit* être en majuscules. Ceci afin que le chaînage de cette méthode ne soit appelé qu'une fois dans son instanciation, au lieu d'être exécuté à chaque fois qu'un en-tête de colonne est modifié.

```
DATA column TYPE REF TO cl_salv_column.
column = alv->get_columns( )->get_column( 'DISTID' ).

column->set_short_text( 'Dist. Unit' ).
column->set_medium_text( 'Unit of Distance' ).
column->set_long_text( 'Mass Unit of Distance (kms, miles)' ).
```

Activer la fonctionnalité de la barre d'outils ALV

L'appel de méthode suivant permet d'utiliser de nombreuses fonctionnalités avancées telles que le tri, le filtrage et l'exportation de données.

```
alv->get_functions( )->set_all( ).
```

Activation de chaque autre rangée de lignes dans ALV

Cette méthode augmente la lisibilité en donnant des lignes consécutives en alternant les nuances de couleur d'arrière-plan.

```
alv->get_display_settings( )->set_striped_pattern( if_salv_c_bool_sap=>true ).
```

Définition du titre d'une ALV affichée

Par défaut, lorsqu'une ALV est affichée, le titre en haut est uniquement le nom du programme. Cette méthode permet à l'utilisateur de définir un titre de 70 caractères maximum. L'exemple suivant montre comment définir un titre dynamique qui affiche le nombre d'enregistrements affichés.

```
alv->get_display_settings( )->set_list_header( |Flight Schedule - { lines( t_spfli ) }
records| ).
```

Lire ABAP GRID List Viewer (ALV) en ligne: https://riptutorial.com/fr/abap/topic/4660/abap-grid-list-viewer--alv-

Chapitre 3: Boucles

Remarques

Lorsque le bouclage sur les tables internes, il est généralement préférable de `ASSIGN` à un symbole de champ plutôt que la boucle `INTO` une zone de travail. L'affectation de symboles de champ met simplement à jour leur référence pour pointer vers la ligne suivante de la table interne pendant chaque itération, tandis que l'utilisation de `INTO` entraîne la copie de la ligne de la table dans la zone de travail, ce qui peut être coûteux pour les tables longues / larges.

Examples

Boucle de table interne

```
LOOP AT itab INTO wa.
ENDLOOP.

FIELD-SYMBOLS <fs> LIKE LINE OF itab.
LOOP AT itab ASSIGNING <fs>.
ENDLOOP.

LOOP AT itab ASSIGNING FIELD-SYMBOL(<fs>).
ENDLOOP.

LOOP AT itab REFERENCE INTO dref.
ENDLOOP.

LOOP AT itab TRANSPORTING NO FIELDS.
ENDLOOP.
```

Bouclage conditionnel

Si seules les lignes correspondant à une certaine condition doivent être insérées dans la boucle, vous pouvez ajouter `WHERE` .

```
LOOP AT itab INTO wa WHERE f1 = 'Max'.
ENDLOOP.
```

En boucle

ABAP propose également le traditionnel `WHILE` -Loop qui s'exécute jusqu'à ce que l'expression donnée soit évaluée à false. Le champ système `sy-index` sera augmenté pour chaque étape de la boucle.

```
WHILE condition.
* do something
ENDWHILE
```

Do Loop

Sans aucun ajout, `DO` -Loop s'exécute sans fin ou du moins jusqu'à ce qu'il soit explicitement sorti de l'intérieur. Le champ système `sy-index` sera augmenté pour chaque étape de la boucle.

```
DO.
* do something... get it?
* call EXIT somewhere
ENDDO.
```

L'ajout de `TIMES` offre un moyen très pratique de répéter le code (le `amount` représente une valeur de type `i`).

```
DO amount TIMES.
* do several times
ENDDO.
```

Commandes Générales

Pour casser des boucles, la commande `EXIT` peut être utilisée.

```
DO.
    READ TABLE itab INDEX sy-index INTO DATA(wa).
    IF sy-subrc <> 0.
        EXIT. "Stop this loop if no element was found
    ENDIF.
    " some code
ENDDO.
```

Pour passer à l'étape suivante, vous pouvez utiliser la commande `CONTINUE` .

```
DO.
    IF sy-index MOD 1 = 0.
        CONTINUE. " continue to next even index
    ENDIF.
    " some code
ENDDO.
```

L'instruction `CHECK` est un `CONTINUE` avec condition. Si la condition s'avère **fausse** , `CONTINUE` sera exécuté. En d'autres termes: *la boucle ne poursuivra l'étape que si la condition est vraie* .

Cet exemple de `CHECK` ...

```
DO.
    " some code
    CHECK sy-index < 10.
    " some code
ENDDO.
```

... est équivalent à ...

```
DO.
    " some code
    IF sy-index >= 10.
        CONTINUE.
    ENDIF.
    " some code
ENDDO.
```

Lire Boucles en ligne: https://riptutorial.com/fr/abap/topic/2270/boucles

Chapitre 4: commentaires

Examples

Fin de ligne

Tout texte suivant un `"` caractère sur la même ligne est commenté:

```
DATA ls_booking TYPE flightb. " Commented text
```

Ligne complète

Le caractère `*` commente une ligne entière. Le `*` doit être le premier caractère de la ligne.

```
* DATA ls_booking TYPE flightb. Nothing on this line will be executed.
```

Lire commentaires en ligne: https://riptutorial.com/fr/abap/topic/1644/commentaires

Chapitre 5: Conventions de nommage

Syntaxe

- Les caractères, les nombres et _ peuvent être utilisés pour le nom de la variable.
- Deux caractères à utiliser pour l'état de la variable et le type d'objet.
- Les variables locales commencent par L.
- Les variables globales commencent par G.
- Le paramètre d'entrée de fonction commence par I (importation).
- Le paramètre de sortie de la fonction commence par E (exportation).
- Le symbole des structures est S.
- Le symbole de la table est T.

Examples

Variable locale

```
data: lv_temp type string.
data: ls_temp type sy.
data: lt_temp type table of sy.
```

Variable globale

```
data: gv_temp type string.
data: gs_temp type sy.
data: gt_temp type table of sy.
```

Lire Conventions de nommage en ligne: https://riptutorial.com/fr/abap/topic/6770/conventions-de-nommage

Chapitre 6: Cordes

Examples

Littéraux

ABAP propose trois opérateurs différents pour déclarer des variables de type chaîne ou char

Des symboles	Type interne	Longueur	prénom
'...'	C	1-255 Chars	littéraux de champ de texte
`...`	C chaîne	0-255 caractères	littéraux de chaîne de texte
| ... |	C chaîne	0-255 caractères	littéraux de modèle

Notez que l'intervalle de longueur ne s'applique qu'aux valeurs codées en dur. Les `CString` CString en interne ont une longueur arbitraire alors que les variables de type `C` ont toujours une longueur fixe.

Modèles de chaîne

Les modèles de chaîne sont un moyen pratique de mélanger des chaînes littérales avec des valeurs provenant de variables:

```
WRITE |Hello, { lv_name }, nice to meet you!|.
```

Il peut aussi formater des choses comme des dates. Pour utiliser le format de date de l'utilisateur connecté:

```
WRITE |The order was completed on { lv_date DATE = USER } and can not be changed|.
```

Les appels et expressions de méthodes fonctionnelles sont pris en charge:

```
WRITE |Your token is { to_upper( lv_token ) }|.
WRITE |Version is: { cond #( when lv_date < sy-datum then 'out of date' else 'up to date' )
}|.
```

> **Attention!** L'implémentation directe de résultats temporaires (comme les appels de méthode) à l'intérieur des modèles de chaîne peut entraîner des problèmes de performances massifs (pour en savoir plus, cliquez ici). Bien que l'utilisation à l'intérieur d'instructions rarement exécutées soit acceptable, votre programme ralentit rapidement dans les boucles.

Chaînes concaténantes

Les variables de type chaîne et caractère peuvent être concaténées à l'aide de la commande ABAP `CONCATENATE` . Une variable supplémentaire pour stocker les résultats est requise.

Exemple:

```
CONCATENATE var1 var2 var3 INTO result.
"result now contains the values of var1, var2 & var3 stringed together without spaces
```

Sténographie

Les nouvelles versions d'ABAP offrent une très courte variante de concaténation à l'aide de && (opérateur de chaînage).

```
DATA(lw_result) = `Sum: ` && lw_sum.
```

> **Attention!** Il est intéressant de noter que l'utilisation de résultats temporaires en combinaison avec l'opérateur de chaînage à l'intérieur des boucles peut entraîner des problèmes de performances considérables dus à l'augmentation des instructions de copie (pour en savoir plus, cliquez ici).

Lire Cordes en ligne: https://riptutorial.com/fr/abap/topic/3531/cordes

Chapitre 7: Déclaration de données

Examples

Déclaration de données en ligne

Dans certaines situations, les déclarations de données peuvent être effectuées en ligne.

```
LOOP AT lt_sflight INTO DATA(ls_sflight).
    WRITE ls_sflight-carrid.
ENDLOOP.
```

Déclaration à variable unique

```
DATA begda TYPE sy-datum.
```

Déclaration de variables multiples

```
DATA: begda TYPE sy-datum,
      endda TYPE sy-datum.
```

Déclaration de données en ligne dans l'instruction SELECT

Lors de l'utilisation d'une déclaration de données en ligne dans un bloc `SELECT...ENDSELECT` ou `SELECT SINGLE` , le caractère `@` doit être utilisé comme caractère d'échappement pour l'expression `DATA(lv_cityto)` . Une fois que le caractère d'échappement a été utilisé, toutes les autres variables hôtes doivent également être échappées (comme c'est le cas avec `lv_carrid` ci-dessous).

```
DATA lv_carrid TYPE s_carr_id VALUE 'LH'.
SELECT SINGLE cityto FROM spfli
       INTO @DATA(lv_cityto)
       WHERE carrid = @lv_carrid
       AND   connid = 2402.
WRITE: / lv_cityto.
```

Sorties `BERLIN` .

Options de déclaration de variable

Différents types de variables peuvent être déclarés avec des options spéciales.

```
DATA: lv_string   TYPE string, " standard declaration
      lv_char     TYPE c,      " declares a character variable of length 1
      lv_char5(5) TYPE c,      " declares a character variable of length 5
      l_packed TYPE p LENGTH 10 DECIMALS 5 VALUE '1234567890.123456789'. " evaluates to
1,234,567,890.12346
```

Lire Déclaration de données en ligne: https://riptutorial.com/fr/abap/topic/1646/declaration-de-donnees

Chapitre 8: Expressions régulières

Examples

Remplacer

L'instruction `REPLACE` peut fonctionner directement avec des expressions régulières:

```
DATA(lv_test) = 'The quick brown fox'.
REPLACE ALL OCCURRENCES OF REGEX '\wo' IN lv_test WITH 'XX'.
```

La variable `lv_test` évaluera à `The quick bXXwn XXx` .

Recherche

L'instruction `FIND` peut fonctionner directement avec des expressions régulières:

```
DATA(lv_test) = 'The quick brown fox'.

FIND REGEX '..ck' IN lv_test.
" sy-subrc == 0

FIND REGEX 'a[sdf]g' IN lv_test.
" sy-subrc == 4
```

Expressions régulières orientées objet

Pour les opérations regex plus avancées, il est préférable d'utiliser `CL_ABAP_REGEX` et ses classes associées.

```
DATA: lv_test  TYPE string,
      lo_regex TYPE REF TO cl_abap_regex.

lv_test = 'The quick brown fox'.
CREATE OBJECT lo_regex
  EXPORTING
    pattern = 'q(...)\w'.

DATA(lo_matcher) = lo_regex->create_matcher( text = lv_test ).
WRITE: / lo_matcher->find_next( ).      " X
WRITE: / lo_matcher->get_submatch( 1 ). " uic
WRITE: / lo_matcher->get_offset( ).     " 4
```

Évaluation des expressions régulières avec une fonction de prédicat

La fonction de prédicat `matches` peut être utilisée pour évaluer des chaînes à la volée sans utiliser de déclaration d'objet.

```
IF matches( val = 'Not a hex string'
```

```
            regex = '[0-9a-f]*' ).
  cl_demo_output=>display( 'This will not display' ).
ELSEIF matches( val = '6c6f7665'
                regex = '[0-9a-f]*' ).
  cl_demo_output=>display( 'This will display' ).
ENDIF.
```

Obtenir des sous-modèles à l'aide des expressions OO-Regular

En utilisant la méthode `GET_SUBMATCH` de la classe `CL_ABAP_MATCHER` , nous pouvons obtenir les données dans les groupes / sous-groupes.

Objectif: récupérer le jeton à droite du mot-clé "Type".

```
DATA: lv_pattern TYPE string VALUE 'type\s+(\w+)',
      lv_test TYPE string VALUE 'data lwa type mara'.

CREATE OBJECT ref_regex
  EXPORTING
        pattern     = lv_pattern
        ignore_case = c_true.

ref_regex->create_matcher(
    EXPORTING
        text    = lv_test
    RECEIVING
        matcher = ref_matcher
       ).

ref_matcher->get_submatch(
            EXPORTING
                 index = 0
            RECEIVING
                 submatch = lv_smatch.
```

La variable résultante `lv_smatch` contient la valeur `MARA` .

Lire Expressions régulières en ligne: https://riptutorial.com/fr/abap/topic/5113/expressions-regulieres

Chapitre 9: Instructions de flux de contrôle

Examples

IF / ELSEIF / ELSE

```
IF lv_foo = 3.
  WRITE: / 'lv_foo is 3'.
ELSEIF lv_foo = 5.
  WRITE: / 'lv_foo is 5'.
ELSE.
  WRITE: / 'lv_foo is neither 3 nor 5'.
ENDIF.
```

CAS

```
CASE lv_foo.
  WHEN 1.
    WRITE: / 'lv_foo is 1'.
  WHEN 2.
    WRITE: / 'lv_foo is 2'.
  WHEN 3.
    WRITE: / 'lv_foo is 3'.
  WHEN OTHERS.
    WRITE: / 'lv_foo is something else'.
ENDCASE
```

VÉRIFIER

`CHECK` est une instruction simple qui évalue une expression logique et quitte le bloc de traitement en cours si elle est fausse.

```
METHOD do_something.
  CHECK iv_input IS NOT INITIAL. "Exits method immediately if iv_input is initial

  "The rest of the method is only executed if iv_input is not initial
ENDMETHOD.
```

AFFIRMER

`ASSERT` est utilisé dans les zones sensibles où vous voulez être absolument sûr qu'une variable a une valeur spécifique. Si la condition logique après `ASSERT` s'avère fausse, une exception non `ASSERTION_FAILED` (`ASSERTION_FAILED`) est `ASSERTION_FAILED` .

```
ASSERT 1 = 1. "No Problem - Program continues

ASSERT 1 = 2. "ERROR
```

`SWITCH` et `COND` offrent une forme particulière de flux de programme conditionnel. Contrairement à `IF` et `CASE` , ils représentent différentes valeurs basées sur une expression plutôt que d'exécuter des instructions. C'est pourquoi ils comptent comme fonctionnels.

COND

Chaque fois que plusieurs conditions doivent être prises en compte, `COND` peut faire le travail. La syntaxe est assez simple:

```
COND <type>(
    WHEN <condition> THEN <value>
    ...
    [ ELSE <default> | throw <exception> ]
).
```

Exemples

```
" Set screen element active depending on radio button
screen-active = COND i(
    WHEN p_radio = abap_true THEN 1
    ELSE 0 " optional, because type 'i' defaults to zero
).

" Check how two operands are related to each other
" COND determines its type from rw_compare
rw_compare = COND #(
    WHEN op1 < op2 THEN 'LT'
    WHEN op1 = op2 THEN 'EQ'
    WHEN op1 > op2 THEN 'GT'
).
```

COMMUTATEUR

`SWITCH` est un outil pratique pour le mappage des valeurs, car il vérifie uniquement l'égalité, donc plus court que `COND` dans certains cas. Si une entrée inattendue a été donnée, il est également possible de lancer une exception. La syntaxe est un peu différente:

```
SWITCH <type>(
    <variable>
    WHEN <value> THEN <new_value>
    ...
    [ ELSE <default> | throw <exception> ]
).
```

Exemples

```
DATA(lw_language) = SWITCH string(
    sy-langu
    WHEN 'E' THEN 'English'
    WHEN 'D' THEN 'German'
    " ...
    ELSE THROW cx_sy_conversion_unknown_langu( )
).
```

Lire Instructions de flux de contrôle en ligne: https://riptutorial.com/fr/abap/topic/7289/instructions-de-flux-de-controle

Chapitre 10: Mot clé Classes de messages / MESSAGE

Introduction

L'instruction `MESSAGE` peut être utilisée pour interrompre le flux du programme afin d'afficher des messages courts à l'utilisateur. Le contenu des messages peut être défini dans le code du programme, dans les symboles de texte du programme ou dans une classe de message indépendante définie dans `SE91` .

Remarques

La longueur maximale d'un message, y compris les paramètres qui lui sont transmis à l'aide de `&` , est de 72 caractères.

Examples

Définition d'une classe de message

```
PROGRAM zprogram MESSAGE-ID sabapdemos.
```

Un message défini par le système peut être stocké dans une classe de message. Le jeton `MESSAGE-ID` définit le `sabapdemos` de la classe de message pour l'ensemble du programme. Si ce n'est pas utilisé, la classe de message doit être spécifiée sur chaque appel `MESSAGE` .

MESSAGE avec symbole de texte prédéfini

```
PROGRAM zprogram MESSAGE-ID za.
...
MESSAGE i000 WITH TEXT-i00.
```

Un message affichera le texte stocké dans le symbole texte `i00` à l'utilisateur. Étant donné que le type de message est i (comme indiqué dans `i000`), une fois que l'utilisateur a quitté la boîte de dialogue, le flux de programme continue à partir du point de l'appel `MESSAGE` .

Bien que le texte ne provienne pas de la classe de message `za` , un `MESSAGE-ID` doit être spécifié.

Message sans classe de message prédéfinie

```
PROGRAM zprogram.
...
MESSAGE i050(sabapdemos).
```

Il peut être gênant de définir une classe de message pour l'ensemble du programme. Il est donc

possible de définir la classe de message dont provient le message dans l'instruction `MESSAGE` elle-même. Cet exemple affichera le message `050` de la classe de message `sabapdemos` .

Messagerie dynamique

```
DATA: msgid TYPE sy-msgid VALUE 'SABAPDEMOS',
      msgty TYPE sy-msgty VALUE 'I',
      msgno TYPE sy-msgno VALUE '050'.

MESSAGE ID mid TYPE mtype NUMBER num.
```

L'appel `MESSAGE` ci-dessus est synonyme de l'appel `MESSAGE i050(sapdemos).` .

Passer des paramètres aux messages

Le symbole `&` peut être utilisé dans un message pour lui permettre de lui transmettre des paramètres.

Paramètres ordonnés

Message `777` de la classe `sabapdemos` :

```
Message with type &1 &2 in event &3
```

L'appel de ce message avec trois paramètres renverra un message en utilisant les paramètres:

```
MESSAGE i050(sabapdemos) WITH 'E' '010' 'START-OF-SELECTION`.
```

Ce message sera affiché sous forme de `Message with type E 010 in event START-OF-SELECTION` . Le numéro à côté du symbole `&` indique l'ordre d'affichage des paramètres.

Paramètres non ordonnés

Message `888` de la classe `sabapdemos` :

```
& & & &
```

L'appel de ce message est similaire:

```
MESSAGE i050(sabapdemos) WITH 'param1' 'param2' 'param3' 'param4'.
```

Cela produira `param1 param2 param3 param4` .

Lire Mot clé Classes de messages / MESSAGE en ligne:
https://riptutorial.com/fr/abap/topic/10691/mot-cle-classes-de-messages---message

Chapitre 11: Objets ABAP

Examples

Déclaration de classe

Les classes ABAP peuvent être déclarées globalement ou localement . Une classe globale peut être utilisée par n'importe quel objet du référentiel ABAP. En revanche, une classe locale ne peut être utilisée que dans la portée déclarée.

```
CLASS lcl_abap_class DEFINITION.
  PUBLIC SECTION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
ENDCLASS.
```

Constructeur, méthodes

Implémentation de classe:

```
CLASS lcl_abap_class DEFINITION.
  PUBLIC SECTION.
    METHODS: constructor,
             method1.
  PROTECTED SECTION.
  PRIVATE SECTION.
    METHODS: method2,
             method3.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
    METHOD constructor.
        "Logic
    ENDMETHOD.

    METHOD method1.
        "Logic
    ENDMETHOD.

    METHOD method2.
        "Logic
        method3( ).
    ENDMETHOD.
```

```
    METHOD method3.
        "Logic
    ENDMETHOD.
ENDCLASS.
```

Exemple d'appel de méthode:

```
DATA lo_abap_class TYPE REF TO lcl_abap_class.
CREATE OBJECT lo_abap_class. "Constructor call
lo_abap_class->method1( ).
```

Méthode avec paramètres (importation, modification, exportation)

Implémentation de classe:

```
CLASS lcl_abap_class DEFINITION.
  PRIVATE SECTION.
    METHODS method1 IMPORTING iv_string TYPE string
                     CHANGING cv_string TYPE string
                    EXPORTING ev_string TYPE string.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
    METHOD method1.
        cv_string = iv_string.
        ev_string = 'example'.
    ENDMETHOD.
ENDCLASS.
```

Exemple d'appel de méthode:

```
method1 (
  EXPORTING iv_string = lv_string
  IMPORTING ev_string = lv_string2
   CHANGING cv_string = lv_string3
).
```

Méthode avec retour du paramètre

Implémentation de classe:

```
CLASS lcl_abap_class DEFINITION.
  PRIVATE SECTION.
    METHODS method1 RETURNING VALUE(rv_string) TYPE string.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
    METHOD method1.
        rv_string = 'returned value'.
    ENDMETHOD.
ENDCLASS.
```

Exemple d'appel de méthode:

```
lv_string = method1( ).
```

Notez que les paramètres déclarés avec `RETURNING` sont transmis par valeur uniquement.

Héritage - définition

Information

L'héritage vous permet de dériver une nouvelle classe d'une classe existante. Vous faites cela en utilisant l' **ajout INHERITING FROM** dans le

Sous- classe **CLASS DEFINITION INHERITING FROM** superclass.

déclaration. La nouvelle sous-classe de classe hérite de tous les composants de la superclasse de classe existante. La nouvelle classe est appelée la sous-classe de la classe dont elle est dérivée. La classe d'origine est appelée la superclasse de la nouvelle classe. Une classe peut avoir plus d'une sous-classe directe, mais elle ne peut avoir qu'une seule superclasse directe.

Implémentation de classe

```
CLASS lcl_vehicle DEFINITION.
ENDCLASS.

CLASS lcl_vehicle IMPLEMENTATION.
ENDCLASS.

CLASS lcl_car DEFINITION INHERITING FROM lcl_vehicle.
ENDCLASS.

CLASS lcl_car IMPLEMENTATION.
ENDCLASS.
```

Héritage - Méthodes et classes abstraites et finales

Information

Les ajouts **ABSTRACT** et **FINAL** aux **méthodes** et aux déclarations **de classe** que vous permettent de définir des méthodes abstraites et finales ou des classes.

Une méthode abstraite est définie dans une classe abstraite et ne peut pas être implémentée dans cette classe. À la place, il est implémenté dans une sous-classe de la classe. Les classes abstraites ne peuvent pas être instanciées.

Une méthode finale ne peut pas être redéfinie dans une sous-classe. Les classes finales ne peuvent pas avoir de sous-classes. Ils concluent un arbre d'héritage.

Implémentation de classe:

```
CLASS lcl_abstract DEFINITION ABSTRACT.
    PUBLIC SECTION.
        METHODS: abstract_method ABSTRACT,
                 final_method FINAL
                 normal_method.

ENDCLASS.

CLASS lcl_abstract IMPLEMENTATION.
    METHOD final_method.
        "This method can't be redefined in child class!
    ENDMETHOD.

    METHOD normal_method.
        "Some logic
    ENDMETHOD.

        "We can't implement abstract_method here!

ENDCLASS.

CLASS lcl_abap_class DEFINITION INHERITING FROM lcl_abstract.
    PUBLIC SECTION.
        METHODS: abstract_method REDEFINITION,
                 abap_class_method.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
    METHOD abstract_method.
        "Abstract method implementation
    ENDMETHOD.

    METHOD abap_class_method.
        "Logic
    ENDMETHOD.
ENDCLASS.
```

Exemple d'appel de méthode:

```
DATA lo_class TYPE REF TO lcl_abap_class.
CREATE OBJECT lo_class.

lo_class->abstract_method( ).
lo_class->normal_method( ).
lo_class->abap_class_method( ).
lo_class->final_method( ).
```

Lire Objets ABAP en ligne: https://riptutorial.com/fr/abap/topic/2244/objets-abap

Chapitre 12: Open SQL

Examples

Instruction SELECT

SELECT est une instruction Open-SQL permettant de lire les données d'une ou plusieurs tables de base de données dans des objets de données .

1. Sélection de tous les enregistrements

```
* This returns all records into internal table lt_mara.
SELECT * FROM mara
         INTO lt_mara.
```

2. Sélection d'un enregistrement unique

```
* This returns single record if table consists multiple records with same key.
SELECT SINGLE * INTO TABLE lt_mara
                FROM mara
                WHERE matnr EQ '400-500'.
```

3. Sélection de notices distinctes

```
* This returns records with distinct values.
SELECT DISTINCT * FROM mara
                  INTO TABLE lt_mara
                  ORDER BY matnr.
```

4. Fonctions d'agrégat

```
* This puts the number of records present in table MARA into the variable lv_var
SELECT COUNT( * ) FROM mara
                  INTO lv_var.
```

Lire Open SQL en ligne: https://riptutorial.com/fr/abap/topic/6885/open-sql

Chapitre 13: Programmation dynamique

Examples

Symboles sur le terrain

Les symboles de champ sont équivalents aux pointeurs, sauf que les symboles de champ sont toujours déréférencés (il n'est pas possible de changer l'adresse en mémoire).

Déclaration

Pour déclarer un champ-symbole, le mot `FIELD-SYMBOLS` clé `FIELD-SYMBOLS` doit être utilisé. Les types peuvent être génériques (`ANY [... TABLE]`) pour traiter une grande variété de variables.

```
FIELD-SYMBOLS: <fs_line>     TYPE any,    "generic
               <fs_struct>   TYPE kna1.   "non-generic
```

Attribuer

Les symboles de champ ne sont pas `unassigned` lors de la déclaration, ce qui signifie qu'ils ne pointent vers rien. L'accès à un symbole de champ non attribué entraînera une exception et, s'il n'est pas capturé, un vidage rapide. Par conséquent, l'état doit être vérifié avec `IS ASSIGNED` :

```
IF <fs> IS ASSIGNED.
*... symbol is assigned
ENDIF.
```

Comme ce ne sont que des références, aucune donnée réelle ne peut être stockée à l'intérieur. Ainsi, les `DATA` déclarées sont nécessaires dans tous les cas d'utilisation.

```
DATA: w_name  TYPE string VALUE `Max`,
      w_index TYPE i      VALUE 1.

FIELD-SYMBOLS <fs_name> TYPE any.

ASSIGN w_name TO <fs_name>. "<fs_name> now gets w_name
<fs_name> = 'Manni'.        "Changes to <fs_name> now also affect w_name

* As <fs_name> is generic, it can also be used for numbers

ASSIGN w_index TO <fs_name>. "<fs_name> now refers to w_index.
ADD 1 TO <fs_name>.          "w_index gets incremented by one
```

Désassigner

Parfois, il peut être utile de réinitialiser un symbole de champ. Cela peut être fait en utilisant `UNASSIGN` .

```
UNASSIGN <fs>.
```

```
* Access on <fs> now leads to an exception again
```

Utiliser pour les tables internes

Les symboles de champ peuvent être utilisés pour modifier des tables internes.

```
LOOP AT itab INTO DATA(wa).
* Only modifies wa_line
    wa-name1 = 'Max'.
ENDLOOP.

LOOP AT itab ASSIGNING FIELD-SYMBOL(<fs>).
* Directly refers to a line of itab and modifies its values
    <fs>-name1 = 'Max'.
ENDLOOP.
```

Attention! Les symboles de champ restent assignés même après avoir quitté la boucle. Si vous souhaitez les réutiliser en toute sécurité, désélectionnez-les immédiatement.

Références de données

L'essentiel pour les références de données est l'ajout de `REF TO` après `TYPE` .

Création dynamique de structures

Si le type d'une structure doit être décidé lors de l'exécution, nous pouvons définir notre structure cible comme référence aux `data` type génériques.

```
DATA wa TYPE REF TO data.
```

Pour donner un type `wa` nous utilisons l'instruction `CREATE DATA` . L'addition `TYPE` peut être spécifiée par:

Référence:

```
CREATE DATA wa TYPE kna1
```

- *Les contrôles statiques sont actifs, il est donc impossible de créer un type inconnu*

Prénom:

```
CREATE DATA wa TYPE (lw_name_as_string)
```

- *Les parenthèses sont nécessaires et `lw_name_as_string` contient le nom des types sous forme de chaîne.*
- *Si le type n'a pas été trouvé, une exception de type `CX_SY_CREATE_DATA_ERROR` sera lancée*

Pour l'instanciation de types créés dynamiquement, l'ajout `HANDLE` peut être spécifié. `HANDLE` reçoit un objet qui hérite de `CL_ABAP_DATADESCR` .

```
CREATE DATA dref TYPE HANDLE obj
```

- `obj` peut être créée en utilisant un **R T** ime **T** ype **S** ervices
- comme `dref` est toujours un datareference, il doit être déréférencé (`->*`) pour être utilisé comme conteneur de données (normalement via les symboles de champ)

RunTime Type Services

RunTime Type Services (*short:* **RTTS**) sont utilisés pour:

- créer des types (RunTime Type Creation; *short:* **RTTC**)
- analyse des types (RunTime Type Identification; *short:* **RTTI**)

Des classes

```
CL_ABAP_TYPEDESCR
  |
  |--CL_ABAP_DATADESCR
  |    |
  |    |--CL_ABAP_ELEMDESCR
  |    |--CL_ABAP_REFDESCR
  |    |--CL_ABAP_COMPLEXDESCR
  |         |
  |         |--CL_ABAP_STRUCTDESCR
  |         |--CL_ABAP_TABLEDESCR
  |
  |--CL_ABAP_OBJECTDESCR
     |
     |--CL_ABAP_CLASSDESCR
     |--CL_ABAP_INTFDESCR
```

`CL_ABAP_TYPEDESCR` est la classe de base. Il implémente les méthodes nécessaires pour décrire:

- `DESCRIBE_BY_DATA`
- `DESCRIBE_BY_NAME`
- `DESCRIBE_BY_OBJECT_REF`
- `DESCRIBE_BY_DATA_REF`

Lire Programmation dynamique en ligne: https://riptutorial.com/fr/abap/topic/4442/programmation-dynamique

Chapitre 14: Programmes de template

Syntaxe

- DÉFINITION DE LA CLASSE ABSTRACT FINAL rend la classe de programme essentiellement statique car les méthodes d'instance ne peuvent jamais être utilisées. L'intention est de garder la classe minimale.

Examples

Programme OO avec des méthodes d'événement essentielles

```
REPORT z_template.

CLASS lcl_program DEFINITION ABSTRACT FINAL.

  PUBLIC SECTION.

    CLASS-METHODS start_of_selection.
    CLASS-METHODS initialization.
    CLASS-METHODS end_of_selection.

ENDCLASS.

CLASS lcl_program IMPLEMENTATION.

  METHOD initialization.

  ENDMETHOD.

  METHOD start_of_selection.

  ENDMETHOD.

  METHOD end_of_selection.

  ENDMETHOD.

ENDCLASS.

INITIALIZATION.

  lcl_program=>initialization( ).

START-OF-SELECTION.

  lcl_program=>start_of_selection( ).

END-OF-SELECTION.

  lcl_program=>end_of_selection( ).
```

Lire Programmes de template en ligne: https://riptutorial.com/fr/abap/topic/10552/programmes-de-

template

Chapitre 15: Tables internes

Examples

Types de tables internes

```
DATA: <TABLE NAME> TYPE <SORTED|STANDARD|HASHED> TABLE OF <TYPE NAME>
      WITH <UNIQUE|NON-UNIQUE> KEY <FIELDS FOR KEY>.
```

Tableau standard

Ce tableau contient toutes les entrées stockées de manière linéaire et les enregistrements sont accessibles de manière linéaire. Pour les grandes tailles de tables, l'accès aux tables peut être lent.

Table triée

Nécessite l'addition `WITH UNIQUE | NON-UNIQUE KEY` . La recherche est rapide en raison d'une recherche binaire. Les entrées ne peuvent pas être ajoutées à cette table car elles risquent de rompre l'ordre de tri. Elles sont donc toujours insérées à l'aide du mot clé `INSERT` .

Table hachée

Nécessite l'addition `WITH UNIQUE | NON-UNIQUE KEY` . Utilise un algorithme de hachage propriétaire pour conserver les paires clé-valeur. Théoriquement, les recherches peuvent être aussi lentes que la table `STANDARD` , mais en pratique, elles sont plus rapides qu'une table `SORTED` prend un temps constant, quelle que soit la taille de la table.

Déclaration des tables internes ABAP

Déclaration de table interne basée sur la définition du type local

```
" Declaration of type
TYPES: BEGIN OF ty_flightb,
         id        TYPE fl_id,
         dat       TYPE fl_date,
         seatno    TYPE fl_seatno,
         firstname TYPE fl_fname,
         lastname  TYPE fl_lname,
         fl_smoke  TYPE fl_smoker,
         classf    TYPE fl_class,
         classb    TYPE fl_class,
         classe    TYPE fl_class,
         meal      TYPE fl_meal,
         service   TYPE fl_service,
```

```
         discout   TYPE fl_discnt,
       END OF lty_flightb.

" Declaration of internal table
DATA t_flightb TYPE STANDARD TABLE OF ty_flightb.
```

Déclaration basée sur la table de base de données

```
DATA t_flightb TYPE STANDARD TABLE OF flightb.
```

Déclaration de tableau interne en ligne

Nécessite la version ABAP> 7.4

```
TYPES t_itab TYPE STANDARD TABLE OF i WITH EMPTY KEY.

DATA(t_inline) = VALUE t_itab( ( 1 ) ( 2 ) ( 3 ) ).
```

Déclaration de table interne avec lignes d'en-tête

Dans ABAP, il y a des tables avec des lignes d'en-tête et des tables sans lignes d'en-tête. Les tableaux comportant des lignes d'en-tête sont un concept plus ancien et ne doivent pas être utilisés dans de nouveaux développements.

Table interne: Table standard avec / sans ligne d'en-tête

Ce code déclare la table `i_compc_all` avec la structure existante de `compc_str` .

```
DATA: i_compc_all TYPE STANDARD TABLE OF compc_str WITH HEADER LINE.
DATA: i_compc_all TYPE STANDARD TABLE OF compc_str.
```

Table interne: Table hachée avec / sans ligne d'en-tête

```
DATA: i_map_rules_c TYPE HASHED TABLE OF /bic/ansdomm0100 WITH HEADER LINE
DATA: i_map_rules_c TYPE HASHED TABLE OF /bic/ansdomm0100
```

Déclaration d'une zone de travail pour les tables sans en-tête

Une zone de travail (généralement abrégée *wa*) a exactement la même structure que la table, mais ne peut contenir qu'une seule ligne (une WA est la structure d'une table avec une seule dimension).

```
DATA: i_compc_all_line LIKE LINE OF i_compc_all.
```

Lire, écrire et insérer dans des tables internes

Lire, écrire et insérer dans des tables internes avec une ligne d'en-tête:

```
" Read from table with header (using a loop):
LOOP AT i_compc_all.              " Loop over table i_compc_all and assign header line
  CASE i_compc_all-ftype.         " Read cell ftype from header line from table i_compc_all
    WHEN 'B'.                     " Bill-to customer number transformation
      i_compc_bil = i_compc_all.  " Assign header line of table i_compc_bil with content of
header line i_compc_all
      APPEND i_compc_bil.         " Insert header line of table i_compc_bil into table
i_compc_bil
    " ... more WHENs
  ENDCASE.
ENDLOOP.
```

> Rappel: les tables internes avec des lignes d'en-tête sont interdites dans les contextes orientés objet. L'utilisation de tables internes *sans* lignes d'en-tête est toujours recommandée.

Lire, écrire et insérer dans des tables internes sans ligne d'en-tête:

```
" Loop over table i_compc_all and assign current line to structure i_compc_all_line
LOOP AT i_compc_all INTO i_compc_all_line.
  CASE i_compc_all_line-ftype.                " Read column ftype from current line (which as
assigned into i_compc_all_line)
    WHEN 'B'.                                 " Bill-to customer number transformation
      i_compc_bil_line = i_compc_all_line.    " Copy structure
      APPEND i_compc_bil_line TO i_compc_bil. " Append structure to table
    " more WHENs ...
  ENDCASE.
ENDLOOP.


" Insert into table with Header:
INSERT TABLE i_sap_knb1.                      " insert into TABLE WITH HEADER: insert table
header into it's content
insert i_sap_knb1_line into table i_sap_knb1. " insert into HASHED TABLE: insert structure
i_sap_knb1_line into hashed table i_sap_knb1
APPEND p_t_errorlog_line to p_t_errorlog.     " insert into STANDARD TABLE: insert structure /
wa p_t_errorlog_line into table p_t_errorlog_line
```

Lire Tables internes en ligne: https://riptutorial.com/fr/abap/topic/1647/tables-internes

Chapitre 16: Tests unitaires

Examples

Structure d'une classe de test

Les classes de test sont créées en tant que classes locales dans un test unitaire spécial.

Ceci est la structure de base d'une classe de test:

```
CLASS lcl_test DEFINITION
               FOR TESTING
               DURATION SHORT
               RISK LEVEL HARMLESS.

PRIVATE SECTION.
  DATA:
    mo_cut TYPE REF TO zcl_dummy.

  METHODS:
    setup,

  "********* 30 chars *********|
  dummy_test                    for testing.
ENDCLASS.

CLASS lcl_test IMPLEMENTATION.
  METHOD setup.
    CREATE OBJECT mo_cut.
  ENDMETHOD.

  METHOD dummy_test.
    cl_aunit_assert=>fail( ).
  ENDMETHOD.
ENDCLASS.
```

Toute méthode déclarée avec `FOR TESTING` sera un test unitaire. `setup` est une méthode spéciale qui est exécutée avant chaque test.

Séparer l'accès aux données de la logique

Un principe important pour les tests unitaires consiste à séparer l'accès aux données de la logique métier. Une technique efficace consiste à définir des interfaces pour l'accès aux données. Votre classe principale utilise toujours une référence à cette interface au lieu de lire ou d'écrire directement des données.

dans le code de production, la classe principale se verra attribuer un objet qui encapsule les accès aux données. Cela pourrait être une instruction select, des appels de fonction mudule, n'importe quoi vraiment. La partie importante est que cette classe ne doit rien faire d'autre. Pas de logique

Lorsque vous testez la classe principale, vous lui attribuez un objet qui sert des données statiques

et factices.

Un exemple pour accéder à la table `SCARR`

Interface d'accès aux données `ZIF_DB_SCARR` :

```
INTERFACE zif_db_scarr
  PUBLIC.
    METHODS get_all
      RETURNING
        VALUE(rt_scarr) TYPE scarr_tab .
ENDINTERFACE.
```

Fausse classe de données et classe de test:

```
CLASS lcl_db_scarr DEFINITION.
  PUBLIC SECTION.
    INTERFACES: zif_db_scarr.
ENDCLASS.

CLASS lcl_db_scarr IMPLEMENTATION.
  METHOD zif_db_scarr~get_all.
    " generate static data here
  ENDMETHOD.
ENDCLASS.

CLASS lcl_test DEFINITION
  FOR TESTING
  DURATION SHORT
  RISK LEVEL HARMLESS.

  PRIVATE SECTION.
    DATA:
      mo_cut TYPE REF TO zcl_main_class.

    METHODS:
      setup.
ENDCLASS.

CLASS lcl_test IMPLEMENTATION.
  METHOD setup.
    DATA: lo_db_scarr TYPE REF TO lcl_db_scarr.

    CREATE OBJECT lo_db_scarr.

    CREATE OBJECT mo_cut
      EXPORTING
        io_db_scarr = lo_db_scarr.
  ENDMETHOD.
ENDCLASS.
```

L'idée ici est que dans le code de production, `ZCL_MAIN_CLASS` obtiendra un objet `ZIF_DB_SCARR` qui effectue un `SELECT` et retourne la table entière alors que le test unitaire s'exécute sur un ensemble de données statique défini directement dans le test unitaire.

Lire Tests unitaires en ligne: https://riptutorial.com/fr/abap/topic/3999/tests-unitaires

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec ABAP	Christian, Community, gkubed, Jagger, mkysoft
2	ABAP GRID List Viewer (ALV)	Achuth hadnoor, gkubed
3	Boucles	Christian, Community, gkubed, Stu G
4	commentaires	4444, Christian, gkubed
5	Conventions de nommage	mkysoft
6	Cordes	Achuth hadnoor, Community, maillard, nexxus, Suncatcher
7	Déclaration de données	Christian, gkubed
8	Expressions régulières	AKHIL RAJ, gkubed, maillard
9	Instructions de flux de contrôle	Community, gkubed, maillard
10	Mot clé Classes de messages / MESSAGE	gkubed
11	Objets ABAP	Community, Michał Majer, Thomas Matecki
12	Open SQL	AKHIL RAJ, gkubed
13	Programmation dynamique	Community, gkubed
14	Programmes de template	nath
15	Tables internes	Community, gkubed, Michał Majer, Rahul Kadukar, Thorsten Niehues
16	Tests unitaires	maillard