ABAP351
Advanced and Generic Programming in ABAP

Gerd Kluger, SAP AG

Björn Mielenhausen, SAP AG

Objectives

After completing this workshop you will be able to:

- **Understand how to make your programs more flexible**
- **Write generic services that can work with arbitrarily structured data**
- **Distinguish different kinds of genericity**
- **Use generic types to add flexibility and safety to ABAP programs**
- **Explain the ABAP type system and the RTTS**
- **Create ABAP types at runtime**

Session Overview

Simple Generic Concepts in ABAP

Dynamic Token Specification

Fully Generic Programs

RTTS & Dynamic Type Creation

Field Symbols

Field symbols are aliases representing fields dynamically

Assignment of fields to field symbols at run time

No copying

Field symbols are not pointers

```
DATA:
    text(20) TYPE c VALUE 'Hello world'.

FIELD-SYMBOLS:
    <fs> TYPE any.

ASSIGN text TO <fs>.
WRITE / <fs>.
```

Casting Field Symbols

```
TYPES: MY_TYPE(9) TYPE C.

DATA:  SmallField(5)  TYPE C,
       LargeField(10) TYPE C VALUE '1234567890',
       TypeName(7)    TYPE C VALUE 'MY_TYPE',
       SomeType       TYPE REF TO cl_abap_typedescr.

FIELD-SYMBOLS: <fa> TYPE ANY,
               <fs> TYPE my_type.

ASSIGN LargeField TO <fs> CASTING.
ASSIGN LargeField TO <fa> CASTING TYPE MY_TYPE.

ASSIGN LargeField TO <fa> CASTING TYPE N.

ASSIGN LargeField TO <fa> CASTING TYPE (TypeName).
SomeType = cl_abap_typedescr=>describe_by_name( 'MY_TYPE' ).
ASSIGN LargeField TO <fa> CASTING TYPE HANDLE SomeType.

ASSIGN LargeField TO <fa> CASTING LIKE SmallField.

ASSIGN LargeField TO <fa> CASTING LIKE <fa>.
```

Casting to ...

... statically completely specified type

... generic type

... dynamically specified type

... static field type

... dynamic field type

Data References (1)

Reference type REF TO *typename* for references to arbitrary data objects

A data reference variable is set by

- `CREATE DATA dref TYPE | LIKE ...`
- `GET REFERENCE OF DataObject INTO dref`

Access to data object of reference

- **Reference is typed?**
  - `X = dref->*.  "access the complete data object`
  - `Y = dref->comp.  "access component of a structure`
- **Reference is untyped? (e.g. REF TO DATA)**
  - `ASSIGN dref->* TO <f>. "access complete data object ...`
  - `ASSIGN COMPONENT 'comp' OF STRUCTURE <f> to <fc>.`
    `"... then access component if data object is a structure`

Data References (2)

References ...

- **... are some kind of *Save Pointers***
- **... can be used as containers for arbitrary data objects, e.g. tables of data references**
- **... can be defined in the data dictionary**

Dynamic Creation of Data Objects (1)

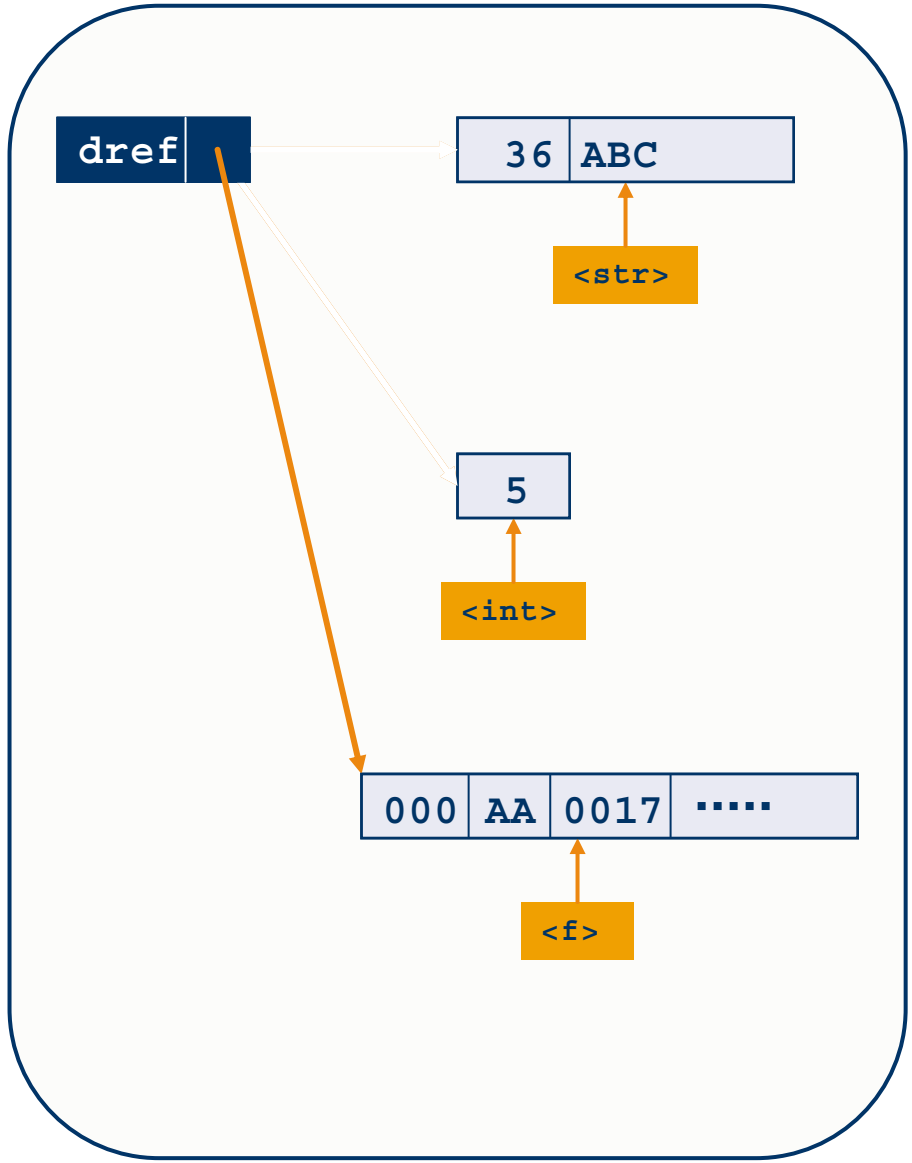
Dynamic instantiation of data types on the heap, for generic programming

Syntax

```
CREATE DATA dref TYPE type | (typename).
CREATE DATA dref TYPE TABLE OF type | (typename).
CREATE DATA dref TYPE REF TO type | (typename).
CREATE DATA dref TYPE HANDLE typeobj.
CREATE DATA dref LIKE field.
```

Dynamic Creation of Data Objects (2)

```
TYPES: BEGIN OF struc,
         a TYPE i,
         b TYPE c LENGTH 8,
       END OF STRUC.

DATA: dref TYPE REF TO DATA,
      tname TYPE string,
      str  TYPE struc,
      int  TYPE i.

FIELD-SYMBOLS: <int> TYPE i,
               <str> TYPE struc,
               <f>   TYPE any.

CREATE DATA dref TYPE struc.
ASSIGN dref->* TO <str>.
<str>-a = 36. <str>-b = 'ABC'.

CREATE DATA dref LIKE int.
ASSIGN dref->* TO <int>.
<int> = 5.

tname = 'SFLIGHT'.
CREATE DATA dref TYPE (tname).
ASSIGN dref->* TO <f>.
SELECT SINGLE * FROM (tname) INTO <f>.
```



Generic Types

Generic type specification of

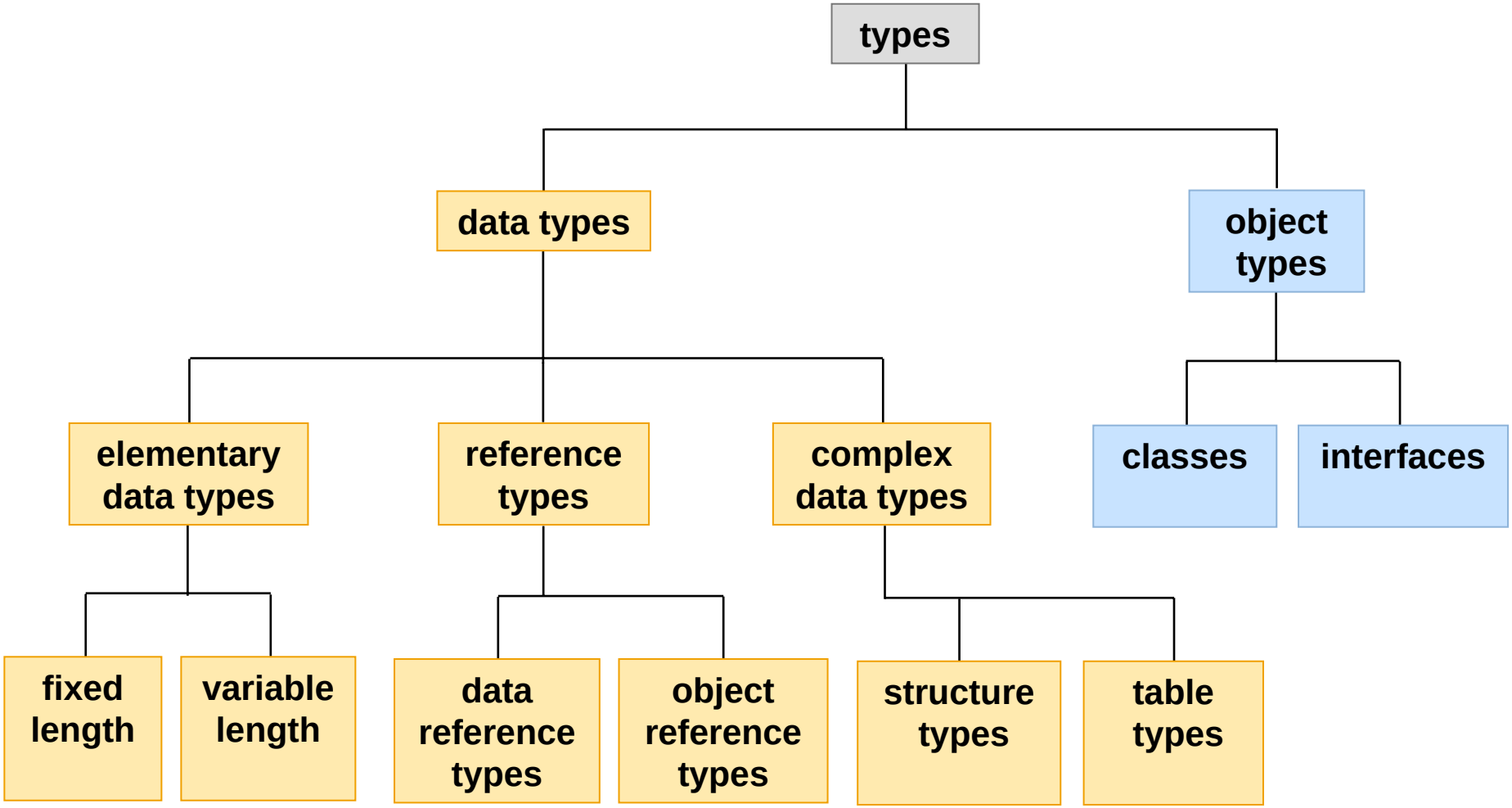
Field Symbols

```
FIELD-SYMBOLS:
    <fs_any> TYPE ANY,
    <fs_c>   TYPE C. "any length
```

Parameters (of subroutines)

```
FORM Foo_1 USING p1 TYPE ANY.
 ...
ENDFORM.

FORM Foo_2 USING p1 TYPE X.
 ...
ENDFORM.
```

ABAP Type Hierarchy



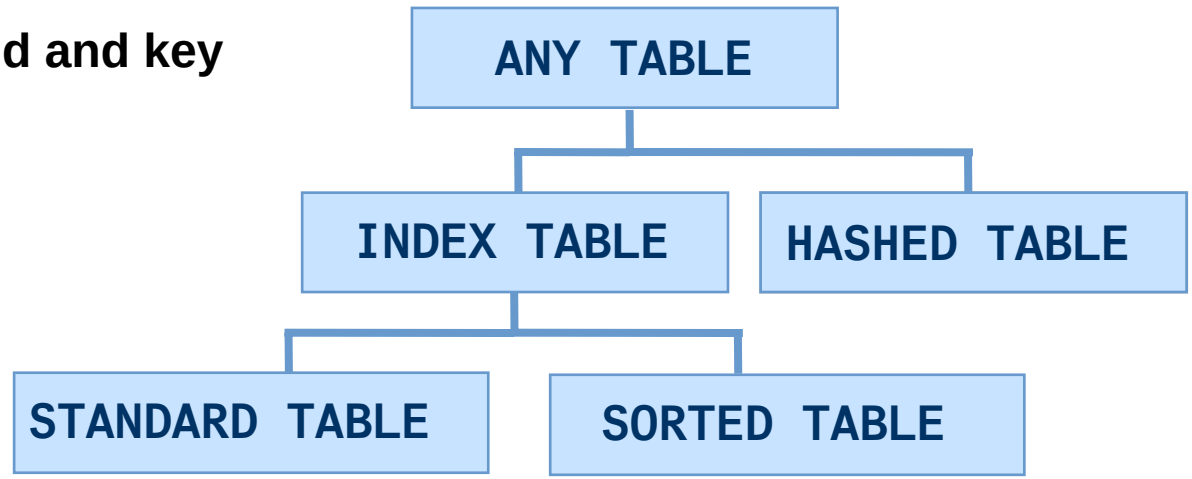
Generic Types

Fully generic — ANY, DATA

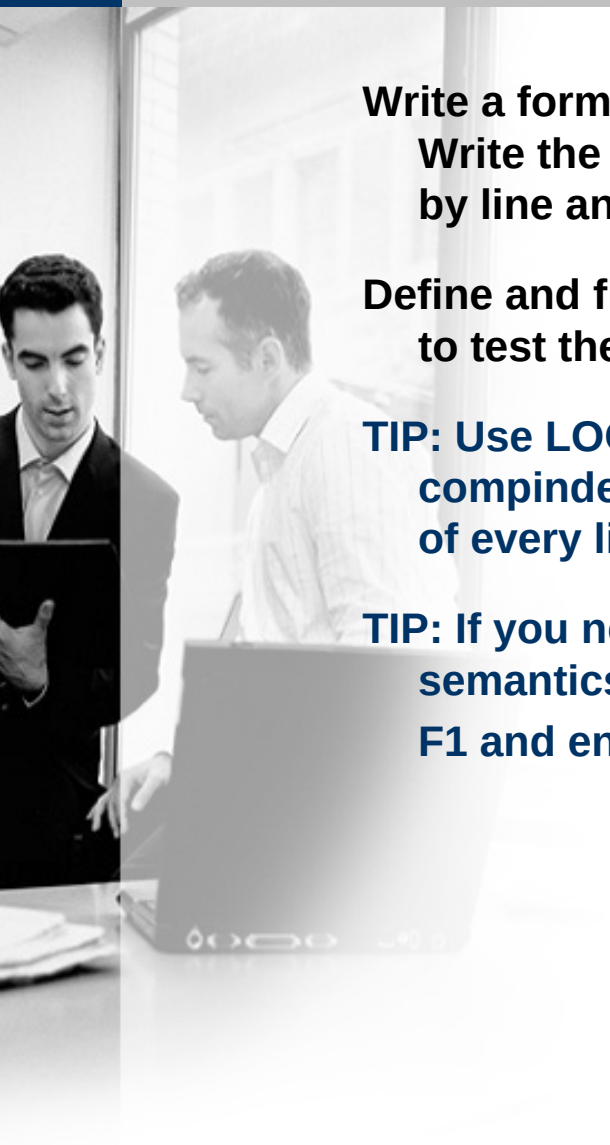
Partially generic — SIMPLE, NUMERIC

Generic length — C, N, X, P
CSEQUENCE, XSEQUENCE, CLIKE

Generic table kind and key



Exercise: Internal Table List Writer



Write a form WRITE_TABLE which accepts any internal table. Write the contents of the internal table to the ABAP list, line by line and field by field.

Define and fill an internal table with a line type of your choice to test the form WRITE_TABLE.

TIP: Use LOOP ASSIGNING nested with ASSIGN COMPONENT compindex OF STRUCTURE and DO loop to access each field of every line.

TIP: If you need more information about the syntax and the semantics of an ABAP statement use Online-Help by pressing F1 and entering the first word of the statement.

Session Overview

Simple Generic Concepts in ABAP

Dynamic Token Specification

Fully Generic Programs

RTTS & Dynamic Type Creation

Dynamic Token Specification

ABAP statements allow to specify some parts dynamically.

General syntax: “(token)” where token is a field evaluated at run time.

```
* Static SORT statement
SORT itab BY comp.

* Dynamic SORT statement
name = 'COMP'.
...
SORT itab BY (name).
```

Rule: The contents of the token must be in upper case.

No static type checks for dynamic statements.

Run time errors occur if the contents of the token are invalid.

Different Types of Dynamic Token Specification

There are 5 types of dynamic token specification

- **Dynamic field specification:**
  The token contains the name of a field
- **Dynamic type specification:**
  The token contains a type name
- **Dynamic component specification:**
  The token contains the name of a component of a structure
- **Dynamic clause specification:**
  The token is an internal table which represents a list of tokens to be inserted and interpreted at run time
- **Dynamic subroutine specification:**
  The token contains the name of a form, method, function, program, ...

Dynamic Field Specification

The token contains the name of a field or database table

- **ASSIGN (field) TO ...**
- **SELECT ... FROM (dbtab) ...**
- **DELETE ... FROM (dbtab) ...**
- **MODIFY (dbtab) ...**
- **UPDATE (dbtab) ...**
- **WRITE ... TO (field)**
- **WRITE (field) TO ...**

```
CONSTANTS: a TYPE i VALUE 1,
           b TYPE i VALUE 2.

DATA: name(5) TYPE c.

FIELD-SYMBOLS: <i> TYPE i.

* Dynamic ASSIGN to constant a
name = 'A'.
ASSIGN (name) TO <i>.
WRITE: <i>.              "=1

* Dynamic ASSIGN to constant b
name = 'B'.
ASSIGN (name) TO <i>.
WRITE: <i>.              "=2
```

The dynamic ASSIGN enables parameters being passed dynamically to arbitrary ABAP statements

Dynamic Type Specification

The token contains the name of a dictionary (global) or an internal type

- **ASSIGN ... CASTING TYPE (type)**
- **CREATE DATA ... TYPE (type) ...**

```
TYPES: BEGIN OF struc,
         a TYPE i,
         b TYPE p,
       END OF struc.
DATA: dref TYPE REF TO DATA.
FIELD-SYMBOLS: <dobj> TYPE any.

PERFORM doSomething USING dref 'SFLIGHT'.
ASSIGN dref->* to <dobj>.
...
FORM doSomething USING dref  TYPE REF TO DATA
                       tname TYPE string.
* Create a data object of type 'tname'
  CREATE DATA dref TYPE (tname).
  ...
ENDFORM.
```

Dynamic Component Specification

The token contains the name of a component of a structure

- **SORT ... BY ($comp_1$) ... ($comp_n$)**
- **READ TABLE ... WITH KEY (k_1) = v_1 ... (k_n) = v_n**
- **DELETE ... COMPARING ($comp_1$) ... ($comp_n$)**
- **MODIFY ... TRANSPORTING ($comp_1$) ... ($comp_n$)**
- **AT NEW/END OF (comp)**
- **ASSIGN COMPONENT comp OF STRUCTURE ...**

Empty tokens are ignored in statements for internal tables

```
DATA: itab TYPE TABLE OF ...,
      key1 TYPE string,
      key2 TYPE string, ...
* Sort table dynamically to set up a dynamic READ with binary search
SORT itab BY (key1) (key2).
READ TABLE itab INTO wa WITH KEY (key1) = val1 (key2) = val2
                        BINARY SEARCH.
```

Dynamic Clause Specification

Dynamic clause represents a sequence of tokens

The syntax of the clause is checked at run time

- **SELECT (fieldlist) ...**
- **SELECT ... GROUP BY (fieldlist)**
- **SELECT ... WHERE (condlist)**

```
TYPES: cond(72) TYPE c.

DATA: wa      TYPE spfli,
      condtab TYPE TABLE OF cond.

* Fill the condition table
APPEND 'CARRID = ''LH'' AND'    TO condtab.
APPEND 'CITYTO = ''NEW YORK''.' TO condtab.

* Database fetch with a dynamic WHERE condition
SELECT * FROM spfli INTO wa WHERE (condtab).
  WRITE: / wa-carrid, wa-connid, wa-cityfrom, wa-cityto.
ENDSELECT.
```

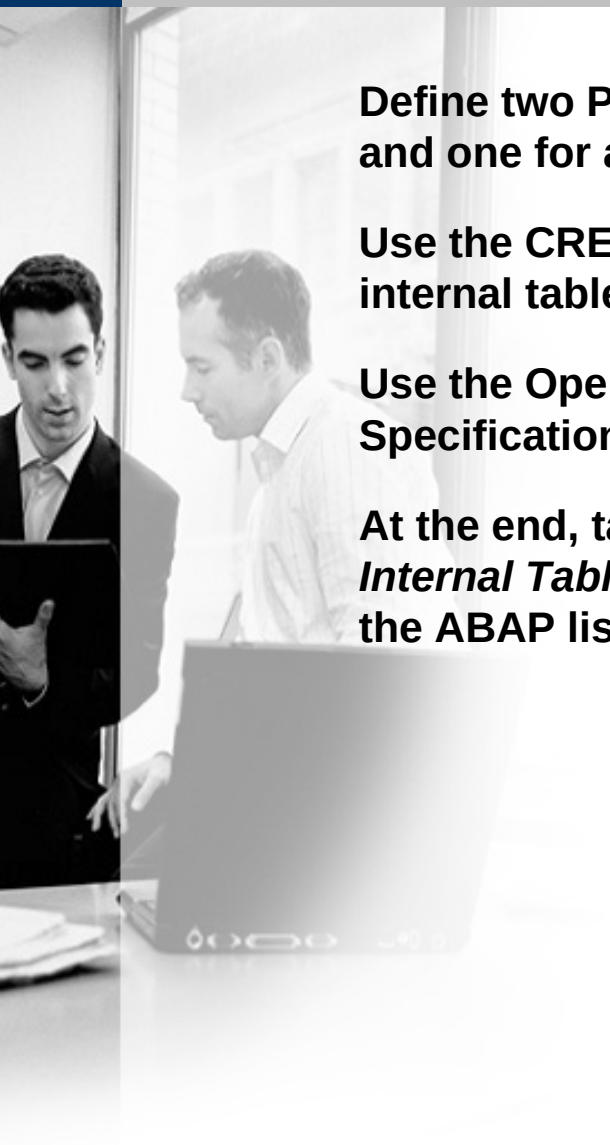
Dynamic Subroutine Specification

The token contains the name of a form, function, method or program and is interpreted at run time to execute the corresponding program unit

- **PERFORM (form) IN PROGRAM (prog) ...**
- **SUBMIT (program) ...**
- **CALL FUNCTION ... PERFORMING (form) ...**
- **CALL METHOD oref->(method) ...**

```
DATA: pname TYPE string,
      fname TYPE string.

fname = 'DO_SOMETHING'.
pname = 'UTILITIES'.

* External PERFORM
PERFORM (fname) IN PROGRAM (pname) USING 999.
```

Exercise: Database List Writer



Define two PARAMETERs, one for a name of a database table and one for a WHERE-Condition.

Use the CREATE DATA statement to create an appropriate internal table as a destination for a SELECT on the database table.

Use the OpenSQL SELECT together with Dynamic Token Specification for the FROM and WHERE clause.

At the end, take the form WRITE_TABLE from „*Exercise: Generic Internal Table List Writer*“ to write the result of the SELECT to the ABAP list.

Session Overview

Simple Generic Concepts in ABAP

Dynamic Token Specification

Fully Generic Programs

RTTS & Dynamic Type Creation

Problem Description

Handle data structures, whose type is completely unknown at design time (dynamic token specification is not sufficient)

Generic tools

- **ALV**
- **DDIC**
- **Data Browser**

Structured data from the *outside* world

- **IMPORT**
- **RFC**
- **IDOC**

Workarounds (1)

Superset approach

- Combination of all possible types in one structure / table
- Use only columns that are actually needed

Drawbacks

- Huge memory overhead

Consequences

- Only useful for structures with small variations or few components

Workarounds (2)

DDIC approach

- **Create DDIC structures or database views dynamically by calling certain function modules**

Drawbacks

- **Structures are persistent**
- **Structure management necessary**
- **Performance**

Consequences

- **Only useful for *long living* structures**

Workarounds (3)

Program generation

- Generate coding either as report or subroutine pool

Drawbacks

- Reports persistent (=> management necessary)
- Limited number of subroutine pools (36)
- Programming cumbersome and error-prone
- Expensive due to compilation

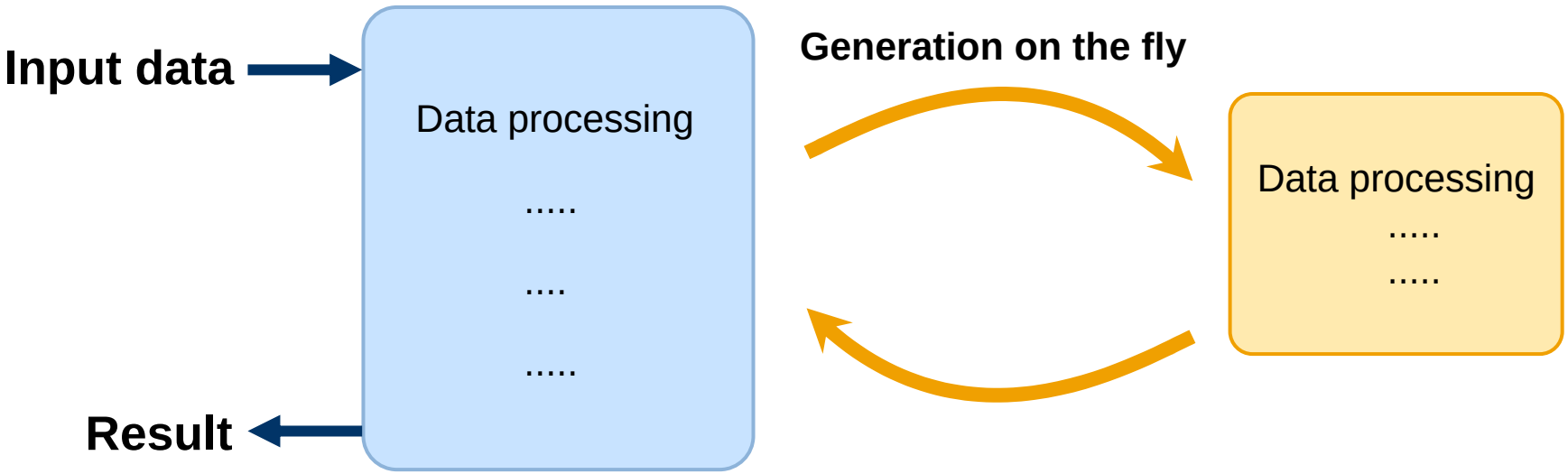
Consequences

- Only useful for coding that rarely changes but is heavily used

Transient Program Generation

Generating transient programs

- More or less frequent change of dynamic input data
- Code generation of a subroutine pool
- Only accessible for internal mode



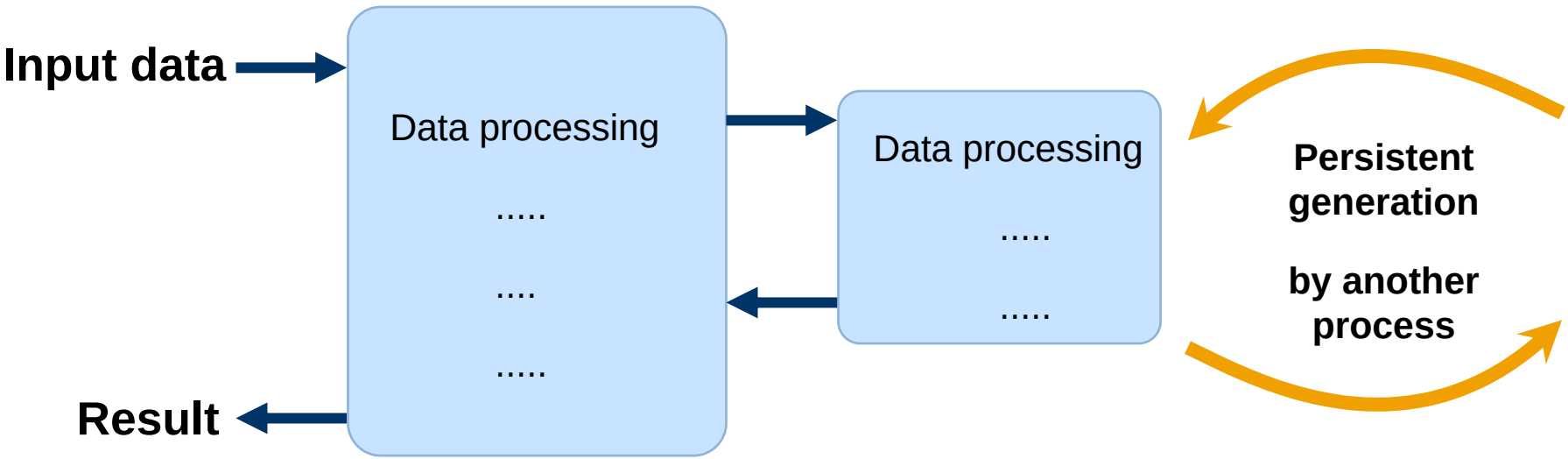
How To Generate Transient Programs

```
DATA: code     TYPE TABLE OF string,
      prog     TYPE program,
      msg(120) TYPE c,
      lin(10)  TYPE c,
      wrd(10)  TYPE c,
      off(3)   TYPE c.

APPEND 'PROGRAM SUBPOOL.' TO code.
APPEND 'FORM DYN1.' TO code.
APPEND 'WRITE / ''Hello, I am a temporary subroutine!''.' TO code.
APPEND 'ENDFORM.' TO code.

GENERATE SUBROUTINE POOL code NAME prog MESSAGE msg
      LINE lin WORD wrd OFFSET off.
IF sy-subrc <> 0.
  WRITE: / 'Error during generation in line', lin,
         / msg, / 'Word:', wrd, 'at offset', off.
ENDIF.

PERFORM dyn1 IN PROGRAM (prog).
```

Persistent Program Generation

Generating persistent programs

- Rare change of dynamic input data
- Code generation can be done by separate process
- Global access



How To Generate Persistent Programs

```
DATA:
   code TYPE TABLE OF string.

CONSTANTS:
   rep(40) VALUE 'ZDYN1'.

APPEND  'PROGRAM ZDYN1.' TO code.
APPEND  'WRITE / ''Hello, I am dynamically created!''.' TO code.
INSERT REPORT rep FROM code.

SUBMIT (rep) AND RETURN.

READ REPORT rep INTO code.
APPEND  'WRITE / ''and I am a dynamic extension!''.' TO code.
INSERT REPORT rep FROM code.

GENERATE REPORT rep.

SUBMIT (rep) AND RETURN.
```

Session Overview

Simple Generic Concepts in ABAP

Dynamic Token Specification

Fully Generic Programs

RTTS & Dynamic Type Creation

Run Time Type Services (RTTS)

Functionality

Type identification and description at run time (formerly RTTI)

Dynamic type creation (RTTC)

Implemented as system classes

Concept

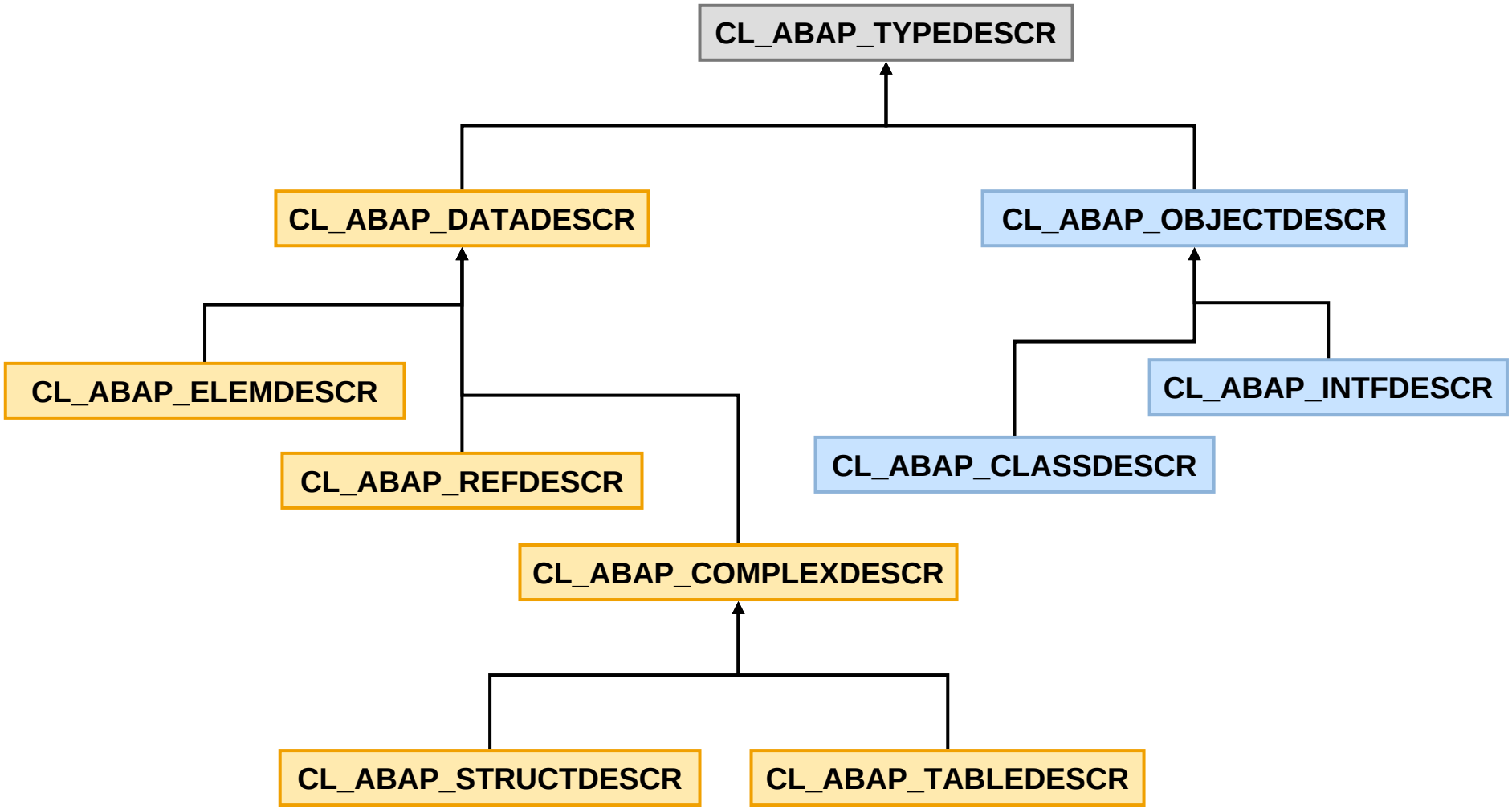
Universal type identification

Each type kind corresponds to one RTTI description class

Type properties represented by attributes

Type creation via factory methods

RTTS Class Hierarchy



Principles of Types in ABAP

Type is well-defined by its type object

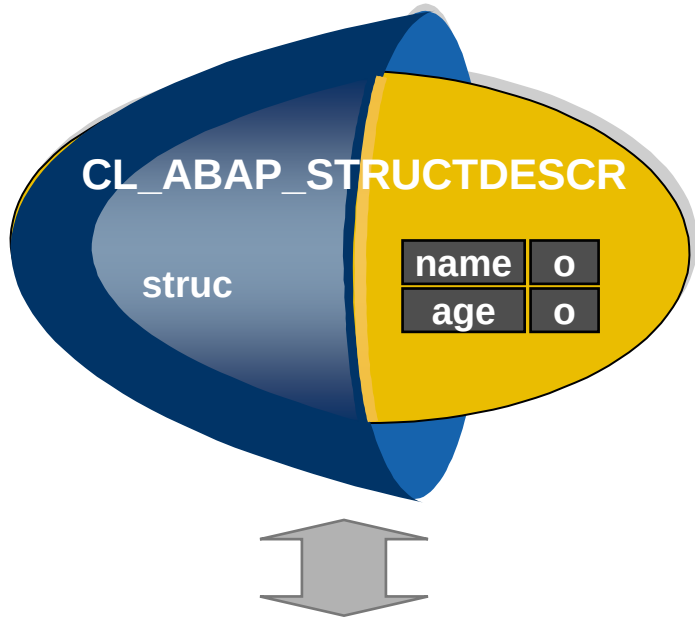
- For every type there is a run time type object
- Every type object corresponds to a type

Type object is instance of RTTS class

Type objects

- Are immutable
- Describe all properties of the type
- Can be used instead of type name

Named types and elementary types are managed by the runtime system




```
TYPES: BEGIN OF struc,
          name TYPE string,
          age  TYPE i,
       END OF struc.
```

How to get a Type Object

Get type object by type name

```
DATA: strucType TYPE REF TO cl_abap_structdescr.
structType ?= cl_abap_typedescr=>describe_by_name( ‘SPFLI’ ).
```

Get type object from a data object

```
DATA: dataType TYPE REF TO cl_abap_datadescr,
      field(5) TYPE C.
dataType ?= cl_abap_typedescr=>describe_by_data( field ).
```

Get elementary types

```
DATA: elemType TYPE REF TO cl_abap_elemdescr.
elemType = cl_abap_elemdescr=>get_i( ).
elemType = cl_abap_elemdescr=>get_c( 20 ).
```

Working with Type Objects

Create a data object of a specific type using a type object

```
DATA: dref    TYPE REF TO DATA,
      c20Type TYPE REF TO cl_abap_elemdescr.
c20Type = cl_abap_elemdescr=>get_c( 20 ).
CREATE DATA dref TYPE HANDLE c20Type.
```

Casting of a field symbol using a type object

```
DATA: x20Type TYPE REF TO cl_abap_elemdescr.
FIELD-SYMBOLS: <fs> TYPE any.
x20Type = cl_abap_elemdescr=>get_x( 20 ).
ASSIGN dref->* TO <fs> CASTING TYPE HANDLE x20Type.
```

Principles of Dynamic Type Creation

Dynamically created types are

- **transient (exist only for the lifetime of the internal mode)**
- **program local (live only in roll area)**
- **anonymous (no name, only accessible by type object)**

Creation of composed data types only

- **table types**
- **reference types**
- **structure types**

Bottom-up approach

- **create new types from existing ones**

Implicit vs. Explicit Type Creation

Implicit type creation

- declarative approach
- only for table and reference types

```
CREATE DATA dref TYPE TABLE OF type.
CREATE DATA dref TYPE REF TO type.
```

Explicit Type creation

- procedural approach
- factory method CREATE( ) in RTTS classes

```
structType = CL_ABAP_STRUCTDESCR=>create( compTab ).
```

Limitations of Dynamically Created Types

Every call of factory method CREATE( ) creates a new type

Types cannot be destroyed

- **type object may be garbage collected, but kernel resources cannot be released (yet)**

Type manager can be useful

```
TYPES: BEGIN OF NamedType,
         name TYPE string,
         type TYPE REF TO CL_ABAP_TYPEDESCR,
       END OF NamedType.
DATA:  typeManager TYPE HASHED TABLE OF NamedType
                            WITH UNIQUE KEY name.
```

Dynamic Creation of Table Types

```
CLASS cl_abap_tabledescr DEFINITION ...
  CLASS-METHODS create
    IMPORTING
      p_line_type  TYPE REF TO cl_abap_datadescr
      p_table_kind TYPE abap_tablekind  DEFAULT tablekind_std
      p_unique     TYPE abap_bool       DEFAULT abap_false
      p_key        TYPE abap_keydescr_tab OPTIONAL
      p_key_kind   TYPE abap_keydefkind DEFAULT keydefkind_default
    RETURNING
      value(p_result) TYPE REF TO cl_abap_tabledescr
    RAISING
      cx_sy_table_creation
```

Line type mandatory

Rest optional

Dynamic Creation of Table Types

```
CLASS cl_abap_tabledescr DEFINITION ...
  CLASS-METHODS create
    IMPORTING
      p_line_type  TYPE REF TO cl_abap_datadescr
      p_table_kind TYPE abap_tablekind  DEFAULT tablekind_std
      p_unique     TYPE abap_bool       DEFAULT abap_false
      p_key        TYPE abap_keydescr_tab OPTIONAL
      p_key_kind   TYPE abap_keydefkind DEFAULT keydefkind_default
    RETURNING
      value(p_result) TYPE REF TO cl_abap_tabledescr
    RAISING
      cx_sy_table_creation
```

alternatives

- tablekind_std
- tablekind_sorted
- tablekind_hashed

Dynamic Creation of Table Types

```
CLASS cl_abap_tabledescr DEFINITION ...
  CLASS-METHODS create
    IMPORTING
      p_line_type  TYPE REF TO cl_abap_datadescr
      p_table_kind TYPE abap_tablekind  DEFAULT tablekind_std
      p_unique     TYPE abap_bool       DEFAULT abap_false
      p_key        TYPE abap_keydescr_tab OPTIONAL
      p_key_kind   TYPE abap_keydefkind DEFAULT keydefkind_default
    RETURNING
      value(p_result) TYPE REF TO cl_abap_tabledescr
    RAISING
      cx_sy_table_creation
```

alternatives

- keydefkind_default
- keydefkind_tableline
- keydefkind_user

Dynamic Creation of Table Types

```
CLASS cl_abap_tabledescr DEFINITION ...
  CLASS-METHODS create
    IMPORTING
      p_line_type  TYPE REF TO cl_abap_datadescr
      p_table_kind TYPE abap_tablekind  DEFAULT tablekind_std
      p_unique     TYPE abap_bool       DEFAULT abap_false
      p_key        TYPE abap_keydescr_tab OPTIONAL
      p_key_kind   TYPE abap_keydefkind DEFAULT keydefkind_default
    RETURNING
      value(p_result) TYPE REF TO cl_abap_tabledescr
    RAISING
      cx_sy_table_creation
```

key fields for structured line types:

alternatives

- keydefkind_default
- keydefkind_tableline
- keydefkind_user

Example of Dynamic Table Type Creation

CARRID	CONNID	DISTANCE	...
LH	0400	6.162	
~~LH~~	~~0400~~	5.347	
QF	0005	1.000	
SQ	0866	1.625	

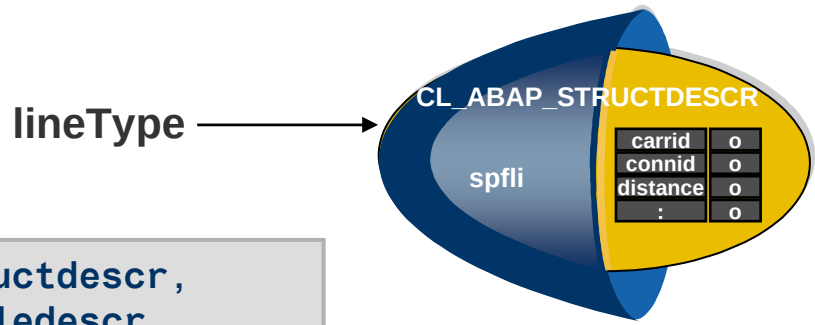
```
TYPES: tableType TYPE SORTED TABLE OF spfli
         WITH UNIQUE KEY carrid connid.
```

```
DATA: lineType  TYPE REF TO cl_abap_structdescr,
      tableType TYPE REF TO cl_abap_tabledescr,
      key       TYPE abap_keydescr_tab.
lineType ?=
  cl_abap_typedescr=>describe_by_name( 'SPFLI' ).
APPEND 'CARRID' TO key. APPEND 'CONNID' TO key.
tableType = cl_abap_tabledescr=>create(
  p_line_type  = lineType
  p_table_kind = cl_abap_tabledescr=>tablekind_sorted
  p_unique     = abap_true
  p_key        = key ).
```

Example of Dynamic Table Type Creation

CARRID	CONNID	DISTANCE	...
LH	0400	6.162	
~~LH~~	~~0400~~	5.347	
QF	0005	1.000	
SQ	0866	1.625	

```
TYPES: tableType TYPE SORTED TABLE OF spfli
         WITH UNIQUE KEY carrid connid.
```



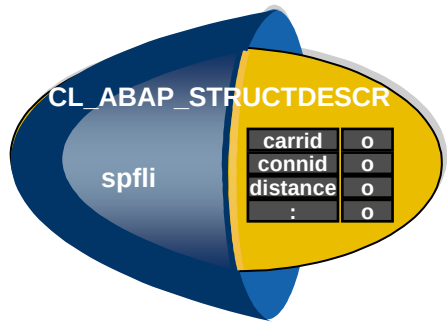

```
DATA: lineType  TYPE REF TO cl_abap_structdescr,
      tableType TYPE REF TO cl_abap_tabledescr,
      key       TYPE abap_keydescr_tab.
lineType ?=
  cl_abap_typedescr=>describe_by_name( 'SPFLI' ).
APPEND 'CARRID' TO key. APPEND 'CONNID' TO key.
tableType = cl_abap_tabledescr=>create(
  p_line_type  = lineType
  p_table_kind = cl_abap_tabledescr=>tablekind_sorted
  p_unique     = abap_true
  p_key        = key ).
```

Example of Dynamic Table Type Creation

CARRID	CONNID	DISTANCE	...
LH	0400	6.162	
~~LH~~	~~0400~~	5.347	
QF	0005	1.000	
SQ	0866	1.625	

```
TYPES: tableType TYPE SORTED TABLE OF spfli
         WITH UNIQUE KEY carrid connid.
```

lineType



keytab
CARRID
CONNID

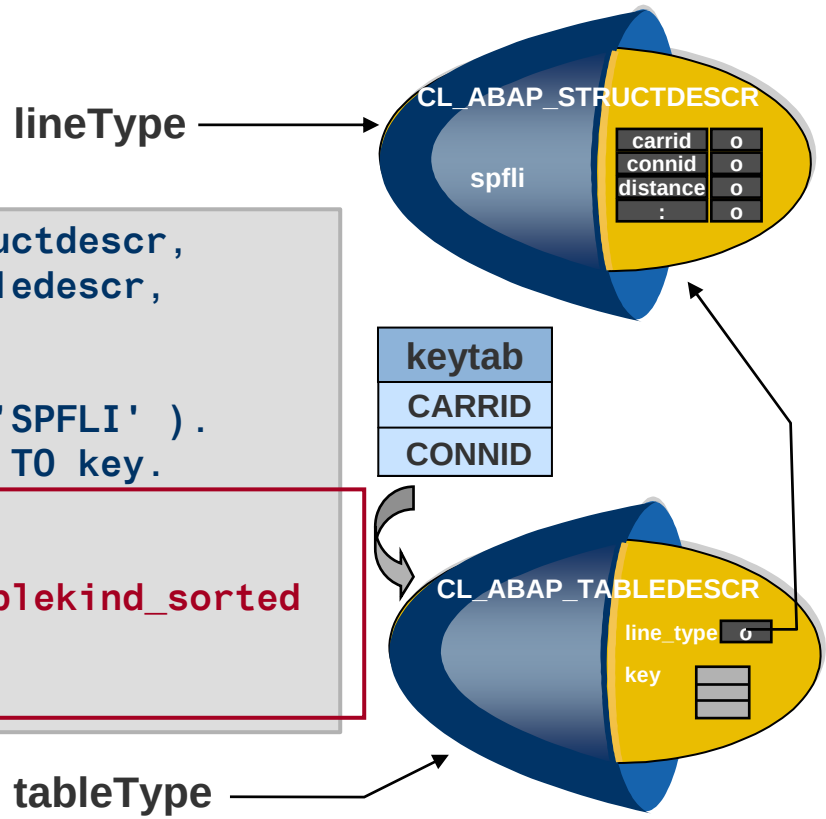
```
DATA: lineType  TYPE REF TO cl_abap_structdescr,
      tableType TYPE REF TO cl_abap_tabledescr,
      key       TYPE abap_keydescr_tab.
lineType ?=
  cl_abap_typedescr=>describe_by_name( 'SPFLI' ).
APPEND 'CARRID' TO key. APPEND 'CONNID' TO key.
tableType = cl_abap_tabledescr=>create(
  p_line_type  = lineType
  p_table_kind = cl_abap_tabledescr=>tablekind_sorted
  p_unique     = abap_true
  p_key        = key ).
```

Example of Dynamic Table Type Creation

CARRID	CONNID	DISTANCE	...
LH	0400	6.162	
~~LH~~	~~0400~~	5.347	
QF	0005	1.000	
SQ	0866	1.625	

```
TYPES: tableType TYPE SORTED TABLE OF spfli
         WITH UNIQUE KEY carrid connid.
```

```
DATA: lineType  TYPE REF TO cl_abap_structdescr,
      tableType TYPE REF TO cl_abap_tabledescr,
      key       TYPE abap_keydescr_tab.
lineType ?=
  cl_abap_typedescr=>describe_by_name( 'SPFLI' ).
APPEND 'CARRID' TO key. APPEND 'CONNID' TO key.
tableType = cl_abap_tabledescr=>create(
  p_line_type  = lineType
  p_table_kind = cl_abap_tabledescr=>tablekind_sorted
  p_unique     = abap_true
  p_key        = key ).
```



Dynamic Creation of Reference Types

```
CLASS cl_abap_refdescr DEFINITION ...
  CLASS-METHODS create
    IMPORTING p_referenced_type TYPE REF TO cl_abap_typedescr
    RETURNING value(p_result)   TYPE REF TO cl_abap_refdescr
    RAISING   cx_sy_ref_creation.
```

Create a reference type from a base type

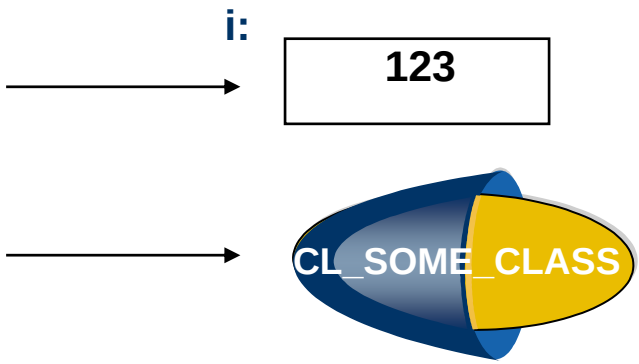
- Base type may be class, interface or data type

Short cut for named base types

- Easier to use
- Much more efficient for Object types

```
  CLASS-METHODS create_by_name
    IMPORTING p_referenced_type_name TYPE csequence
    RETURNING value(p_result) TYPE REF TO cl_abap_refdescr
    RAISING   cx_sy_ref_creation cx_sy_unknown_type.
```

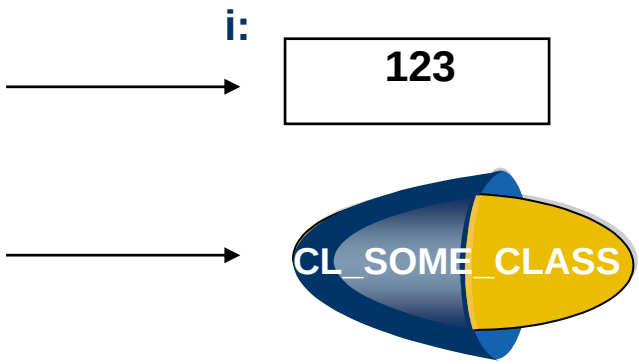
Examples of Dynamic Reference Type Creation



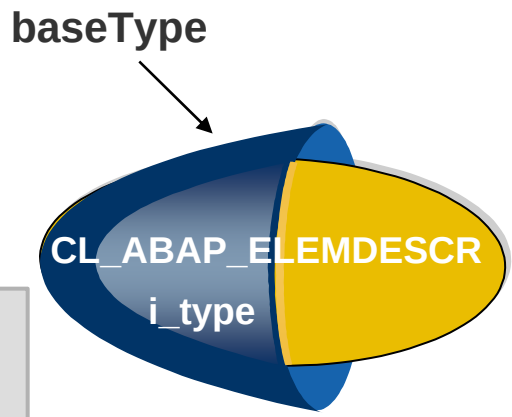

```
TYPES:
  refToIType     TYPE REF TO i,
  refToSomeClass TYPE REF TO cl_some_class.
```

```
DATA: baseType       TYPE REF TO cl_abap_typedescr,
      refToIType     TYPE REF TO cl_abap_refdescr,
      refToSomeClass TYPE REF TO cl_abap_refdescr.

baseType = cl_abap_elemdescr=>get_i( ).
refToIType = cl_abap_refdescr=>create( baseType ).
refToSomeClass =
  cl_abap_refdescr=>create_by_name( 'CL_SOME_CLASS' ).
```

Examples of Dynamic Reference Type Creation

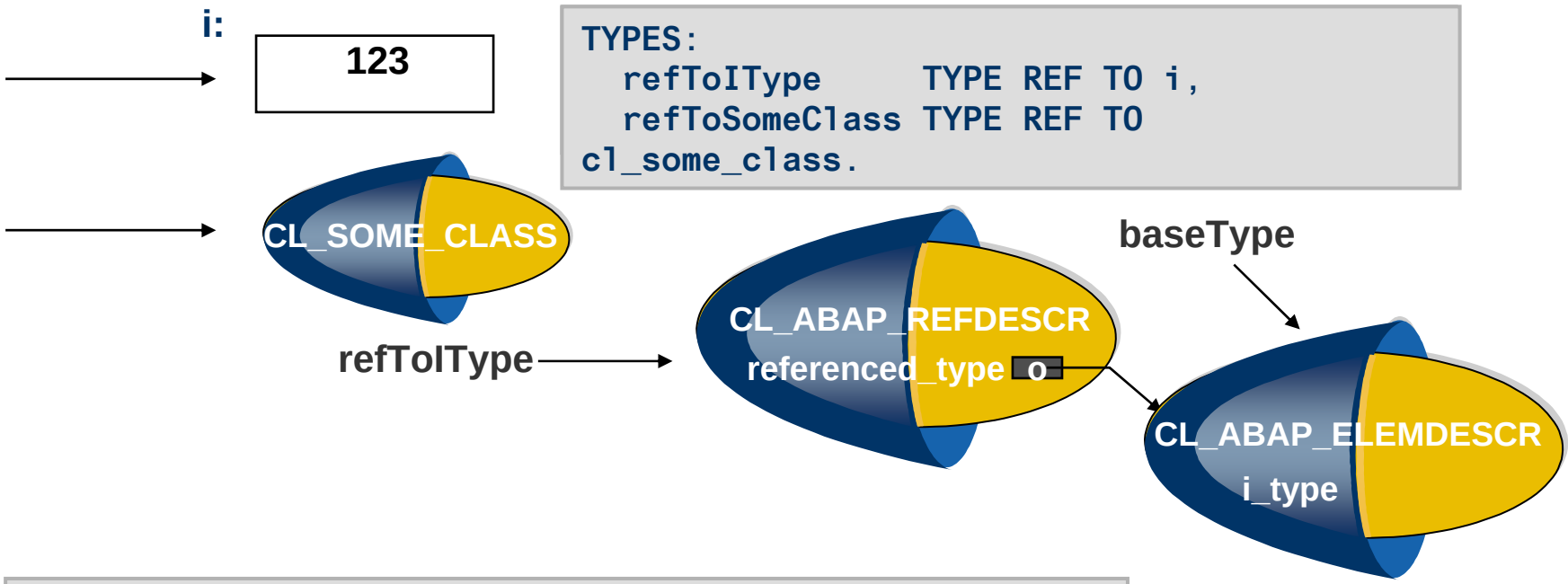



```
TYPES:
  refToIType     TYPE REF TO i,
  refToSomeClass TYPE REF TO
cl_some_class.
```



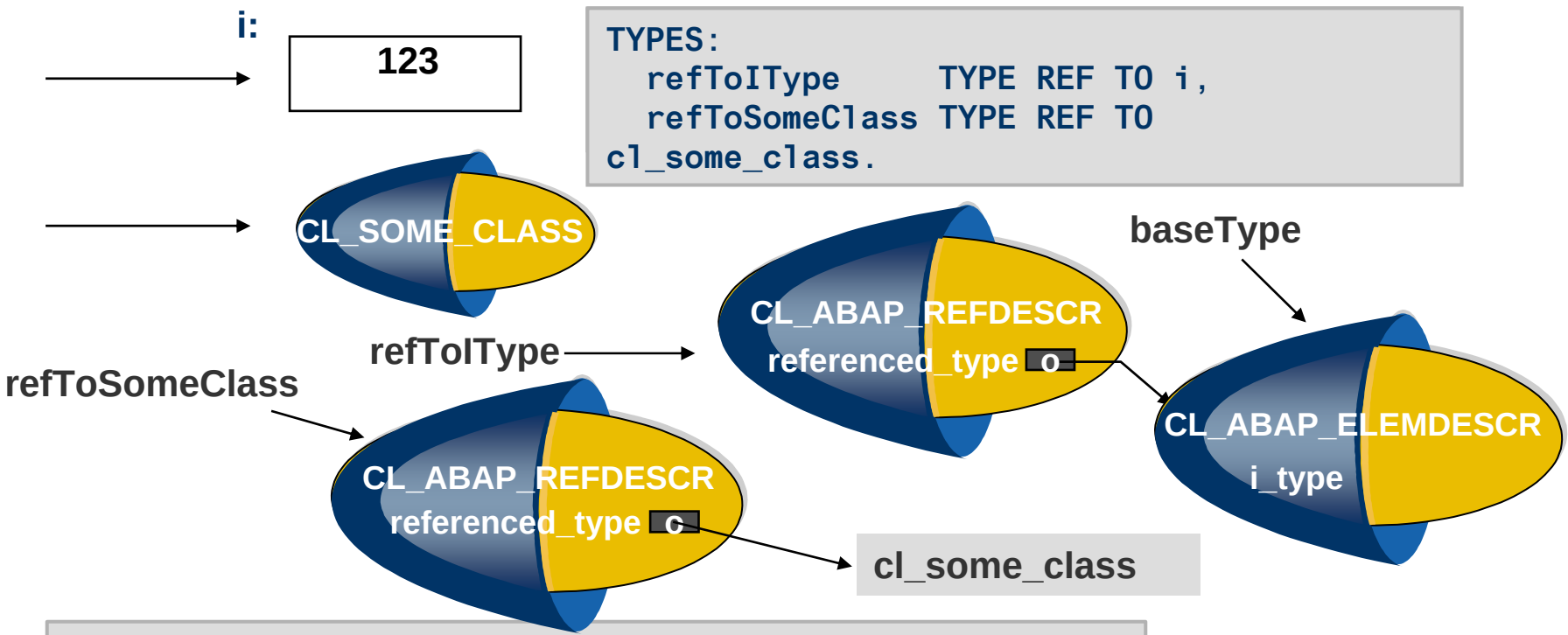

```
DATA: baseType       TYPE REF TO cl_abap_typedescr,
      refToIType     TYPE REF TO cl_abap_refdescr,
      refToSomeClass TYPE REF TO cl_abap_refdescr.

baseType = cl_abap_elemdescr=>get_i( ).
refToIType = cl_abap_refdescr=>create( baseType ).
refToSomeClass =
  cl_abap_refdescr=>create_by_name( 'CL_SOME_CLASS' ).
```

Examples of Dynamic Reference Type Creation




```
DATA: baseType       TYPE REF TO cl_abap_typedescr,
      refToIType     TYPE REF TO cl_abap_refdescr,
      refToSomeClass TYPE REF TO cl_abap_refdescr.

baseType = cl_abap_elemdescr=>get_i( ).
refToIType = cl_abap_refdescr=>create( baseType ).
refToSomeClass =
  cl_abap_refdescr=>create_by_name( 'CL_SOME_CLASS' ).
```

Examples of Dynamic Reference Type Creation




```
TYPES:
  refToIType     TYPE REF TO i,
  refToSomeClass TYPE REF TO
cl_some_class.
```

```
DATA: baseType       TYPE REF TO cl_abap_typedescr,
      refToIType     TYPE REF TO cl_abap_refdescr,
      refToSomeClass TYPE REF TO cl_abap_refdescr.

baseType = cl_abap_elemdescr=>get_i( ).
refToIType = cl_abap_refdescr=>create( baseType ).
refToSomeClass =
  cl_abap_refdescr=>create_by_name( 'CL_SOME_CLASS' ).
```

Dynamic Creation of Structured Types

```
CLASS cl_abap_structdescr DEFINITION ...
  CLASS-METHODS create
    IMPORTING
      p_components    TYPE component_table
      p_strict        TYPE abap_bool  DEFAULT abap_true
    RETURNING
      value(p_result) TYPE REF TO cl_abap_structdescr
    RAISING
      cx_sy_struct_creation.
```

restricts syntax of structure specification to ABAP-OO rules

Create a structured type from a component description table

- component table mandatory
- strictness optional

Component Description Table

NAME	TYPE	AS_INCLUDE	SUFFIX

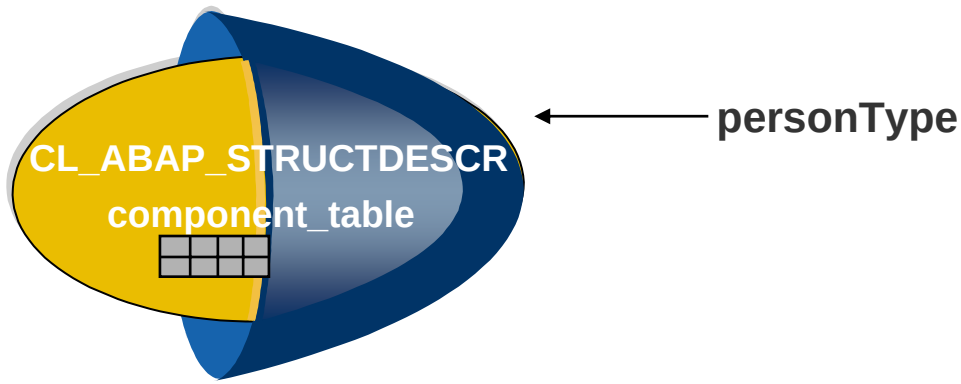
```
TYPES:
  BEGIN OF personType,
    name   TYPE string,
    age(3) TYPE n,
  END OF personType.
```

```
DATA: personType  TYPE REF TO cl_abap_structdescr,
      comp_tab    TYPE cl_abap_structdescr=>component_table.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp_tab = personType->get_components( ).
```

Component Description Table

NAME	TYPE	AS_INCLUDE	SUFFIX

```
TYPES:
  BEGIN OF personType,
    name   TYPE string,
    age(3) TYPE n,
  END OF personType.
```



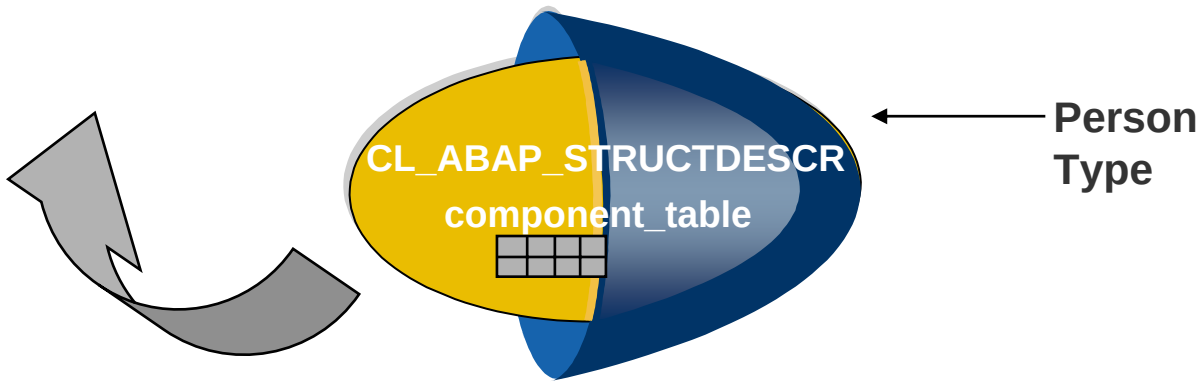

```
DATA: personType  TYPE REF TO cl_abap_structdescr,
      comp_tab    TYPE cl_abap_structdescr=>component_table.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp_tab = personType->get_components( ).
```

Component Description Table

comp_tab:

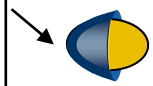
NAME	TYPE	AS_INCLUDE	SUFFIX
NAME	→		
AGE	→		

```
TYPES:
  BEGIN OF personType,
    name   TYPE string,
    age(3) TYPE n,
  END OF personType.
```




```
DATA: personType  TYPE REF TO cl_abap_structdescr,
      comp_tab    TYPE cl_abap_structdescr=>component_table.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp_tab = personType->get_components( ).
```

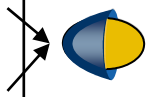
Example of Dynamic Structure Type Creation

NAME	TYPE	AS_INCLUDE	SUFFIX
EMPLOYEE			

```
TYPES:
  BEGIN OF employeeType,
    employee TYPE personType,
    manager  TYPE personType,
  END OF employeeType.
```

```
DATA: personType   TYPE REF TO cl_abap_structdescr,
      employeeType TYPE REF TO cl_abap_structdescr,
      comp_tab     TYPE cl_abap_structdescr=>component_table,
      comp         LIKE LINE OF comp_tab.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp-name = 'EMPLOYEE'. comp-type = personType. append comp to comp_tab.
comp-name = 'MANAGER'.  comp-type = personType. append comp to comp_tab.
employeeType = cl_abap_structdescr=>create( comp_tab ).
```

Example of Dynamic Structure Type Creation

NAME	TYPE	AS_INCLUDE	SUFFIX
EMPLOYEE MANAGER			

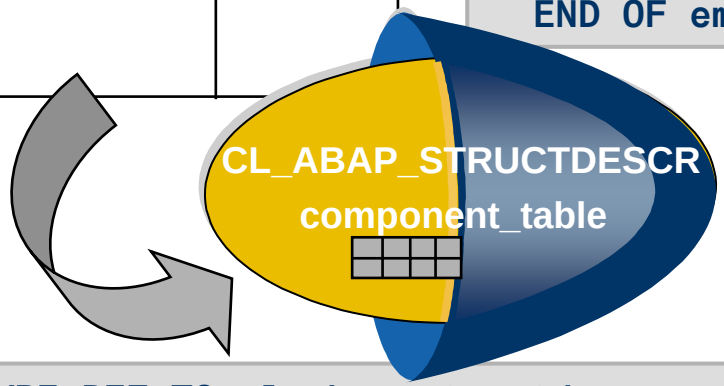
```
TYPES:
  BEGIN OF employeeType,
    employee TYPE personType,
    manager  TYPE personType,
  END OF employeeType.
```

```
DATA: personType   TYPE REF TO cl_abap_structdescr,
      employeeType TYPE REF TO cl_abap_structdescr,
      comp_tab     TYPE cl_abap_structdescr=>component_table,
      comp         LIKE LINE OF comp_tab.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp-name = 'EMPLOYEE'. comp-type = personType. append comp to comp_tab.
comp-name = 'MANAGER'.  comp-type = personType. append comp to comp_tab.
employeeType = cl_abap_structdescr=>create( comp_tab ).
```

Example of Dynamic Structure Type Creation

NAME	TYPE	AS_INCLUDE	SUFFIX
EMPLOYEE			
MANAGER			

```
TYPES:
  BEGIN OF employeeType,
    employee TYPE personType,
    manager  TYPE personType,
  END OF employeeType.
```



employeeType

```
DATA: personType   TYPE REF TO cl_abap_structdescr,
      employeeType TYPE REF TO cl_abap_structdescr,
      comp_tab     TYPE cl_abap_structdescr=>component_table,
      comp         LIKE LINE OF comp_tab.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp-name = 'EMPLOYEE'. comp-type = personType. append comp to comp_tab.
comp-name = 'MANAGER'.  comp-type = personType. append comp to comp_tab.
employeeType = cl_abap_structdescr=>create( comp_tab ).
```

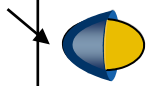
Exercise: Dynamic Table Creation.

Create a new table type which is able to store data records from a database join over tables 'SCARR' and 'SPFLI'. Although both tables have a lot of columns the new table should only contain the columns 'CARRNAME' , 'CONNID', 'CITYFROM', 'CITYTO' and 'DISTANCE'. The new table type should be sorted according to the distance. Once you have created the type, create a data object from it and fill it with all flights that have a distance > 1000.

Tip: First get the component tables of 'SCARR'' and 'SPFLI' and determine the types of the respective columns. Build a new component table from these columns and create the corresponding structure type. Once you have the structure type, create the required table type and dynamically create a data object of this type. Fill this object with an OpenSQL statement that is an inner join on 'SCARR' and 'SPFLI'.

Hint: In order to fill the data object you will need a field symbol (ASSIGN dref->* to <t>).

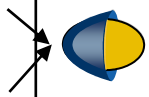
Example of Dynamic Structure with Includes

NAME	TYPE	AS_INCLUDE	SUFFIX
		X	

```
TYPES:  BEGIN OF employeeType.
INCLUDE TYPE personType.
INCLUDE TYPE personType AS manager
        RENAMING WITH SUFFIX _mg.
TYPES:  END OF employeeType.
```

```
DATA: personType   TYPE REF TO cl_abap_structdescr,
      employeeType TYPE REF TO cl_abap_structdescr,
      comp_tab     TYPE cl_abap_structdescr=>component_table,
      comp         LIKE LINE OF comp_tab.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp-name = ''. comp-type = personType.
comp-as_include = abap_true. comp-suffix = ''.
append comp to comp_tab.
comp-name = 'MANAGER'. comp-type = personType.
comp-as_include = abap_true. comp-suffix = '_MG'.
append comp to comp_tab.
employeeType = cl_abap_structdescr=>create( comp_tab ).
```

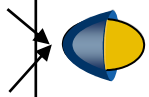
Example of Dynamic Structure with Includes

NAME	TYPE	AS_INCLUDE	SUFFIX
		X	
MANAGER		X	_MG

```
TYPES:  BEGIN OF employeeType.
INCLUDE TYPE personType.
INCLUDE TYPE personType AS manager
        RENAMING WITH SUFFIX _mg.
TYPES:  END OF employeeType.
```

```
DATA: personType   TYPE REF TO cl_abap_structdescr,
      employeeType TYPE REF TO cl_abap_structdescr,
      comp_tab     TYPE cl_abap_structdescr=>component_table,
      comp         LIKE LINE OF comp_tab.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp-name = ''. comp-type = personType.
comp-as_include = abap_true. comp-suffix = ''.
append comp to comp_tab.
comp-name = 'MANAGER'. comp-type = personType.
comp-as_include = abap_true. comp-suffix = '_MG'.
append comp to comp_tab.
employeeType = cl_abap_structdescr=>create( comp_tab ).
```

Example of Dynamic Structure with Includes

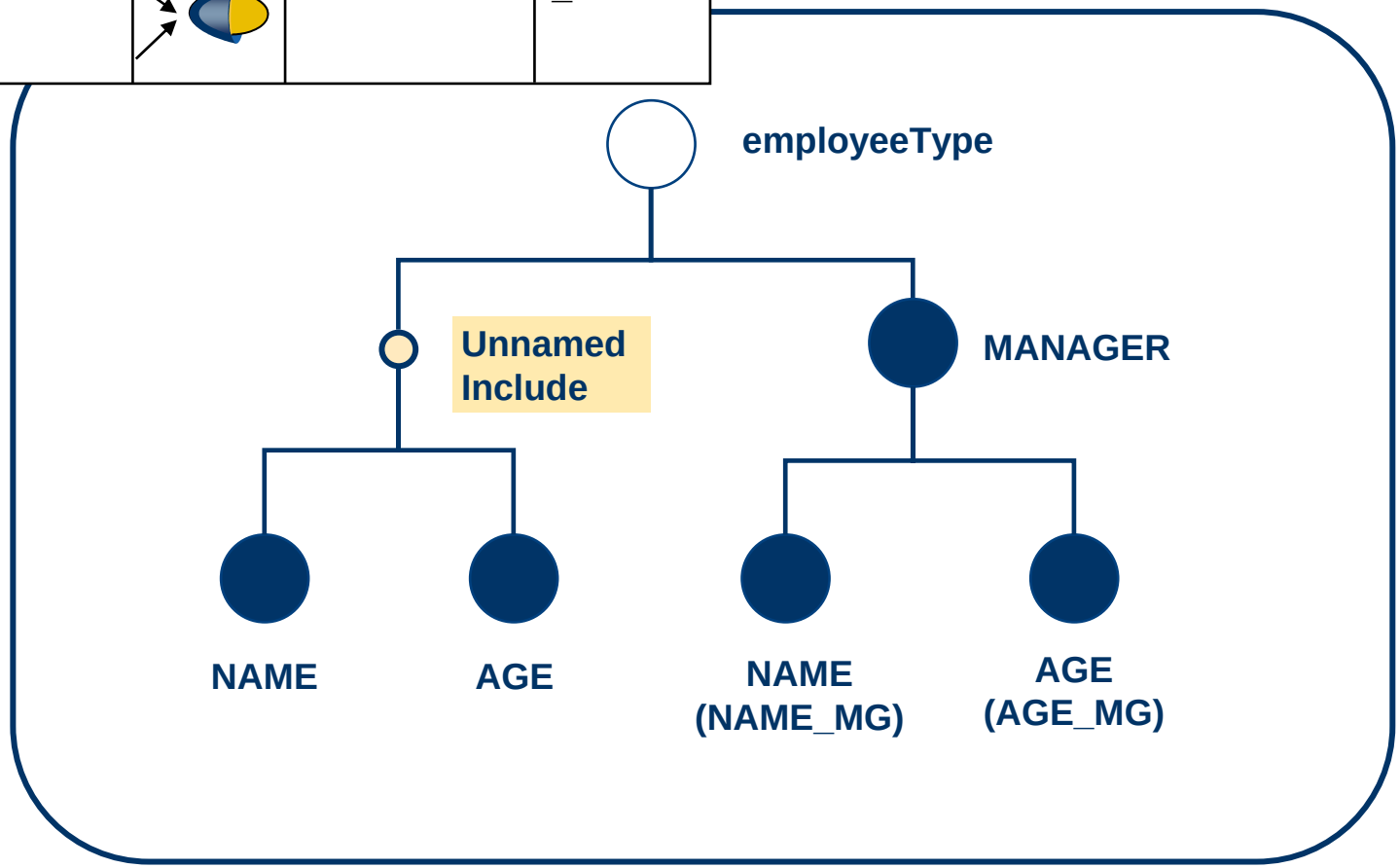
NAME	TYPE	AS_INCLUDE	SUFFIX
		X	
MANAGER		X	_MG

```
TYPES:  BEGIN OF employeeType.
INCLUDE TYPE personType.
INCLUDE TYPE personType AS manager
        RENAMING WITH SUFFIX _mg.
TYPES:  END OF employeeType.
```

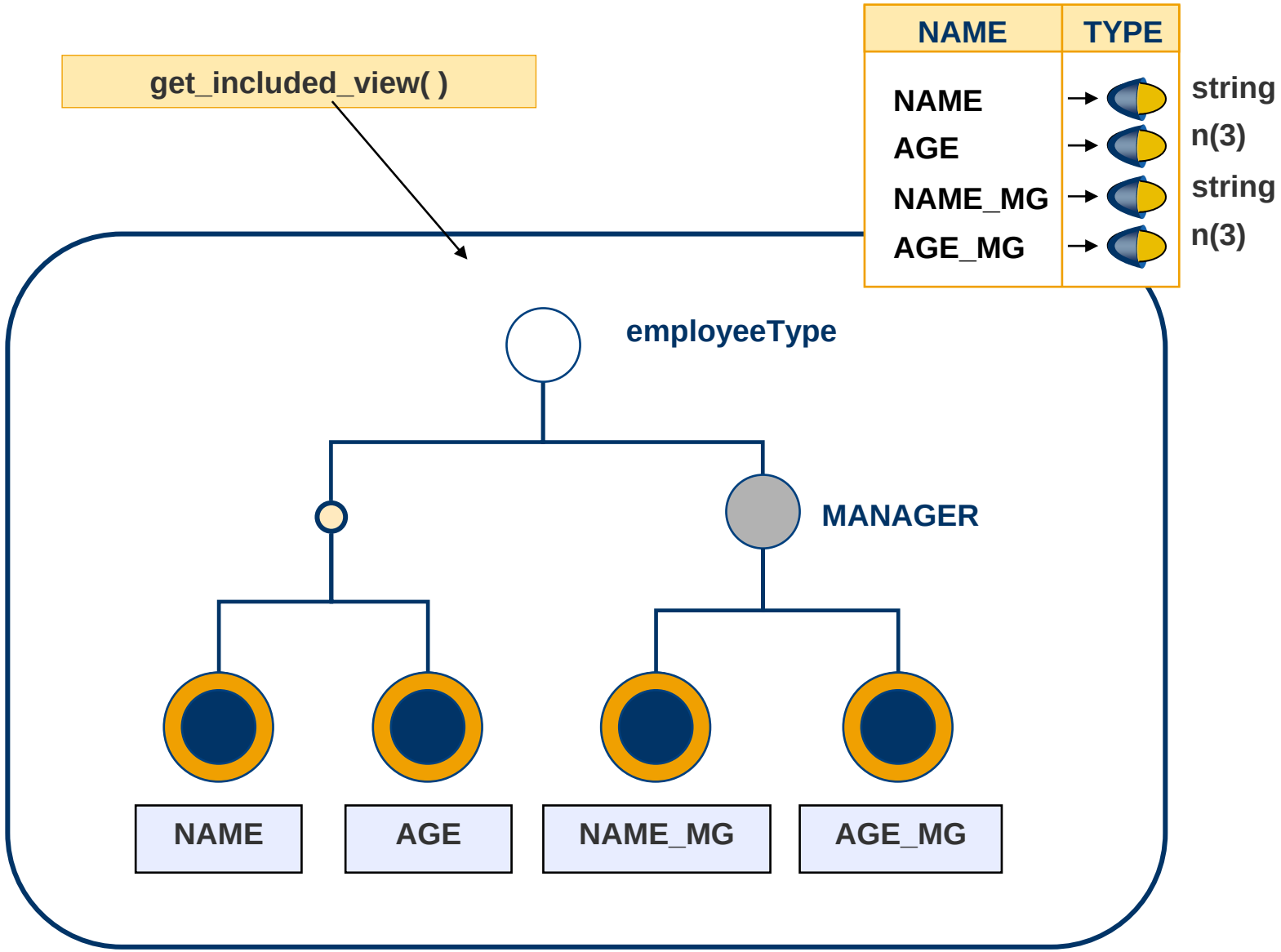
```
DATA: personType   TYPE REF TO cl_abap_structdescr,
      employeeType TYPE REF TO cl_abap_structdescr,
      comp_tab     TYPE cl_abap_structdescr=>component_table,
      comp         LIKE LINE OF comp_tab.

personType ?= cl_abap_typedescr=>describe_by_name( 'personType' ).
comp-name = ''. comp-type = personType.
comp-as_include = abap_true. comp-suffix = ''.
append comp to comp_tab.
comp-name = 'MANAGER'. comp-type = personType.
comp-as_include = abap_true. comp-suffix = '_MG'.
append comp to comp_tab.
employeeType = cl_abap_structdescr=>create( comp_tab ).
```

Analyzing Structures with Includes

NAME	TYPE	AS_INCLUDE	SUFFIX
	personType	X	
MANAGER		X	_MG



Included Views

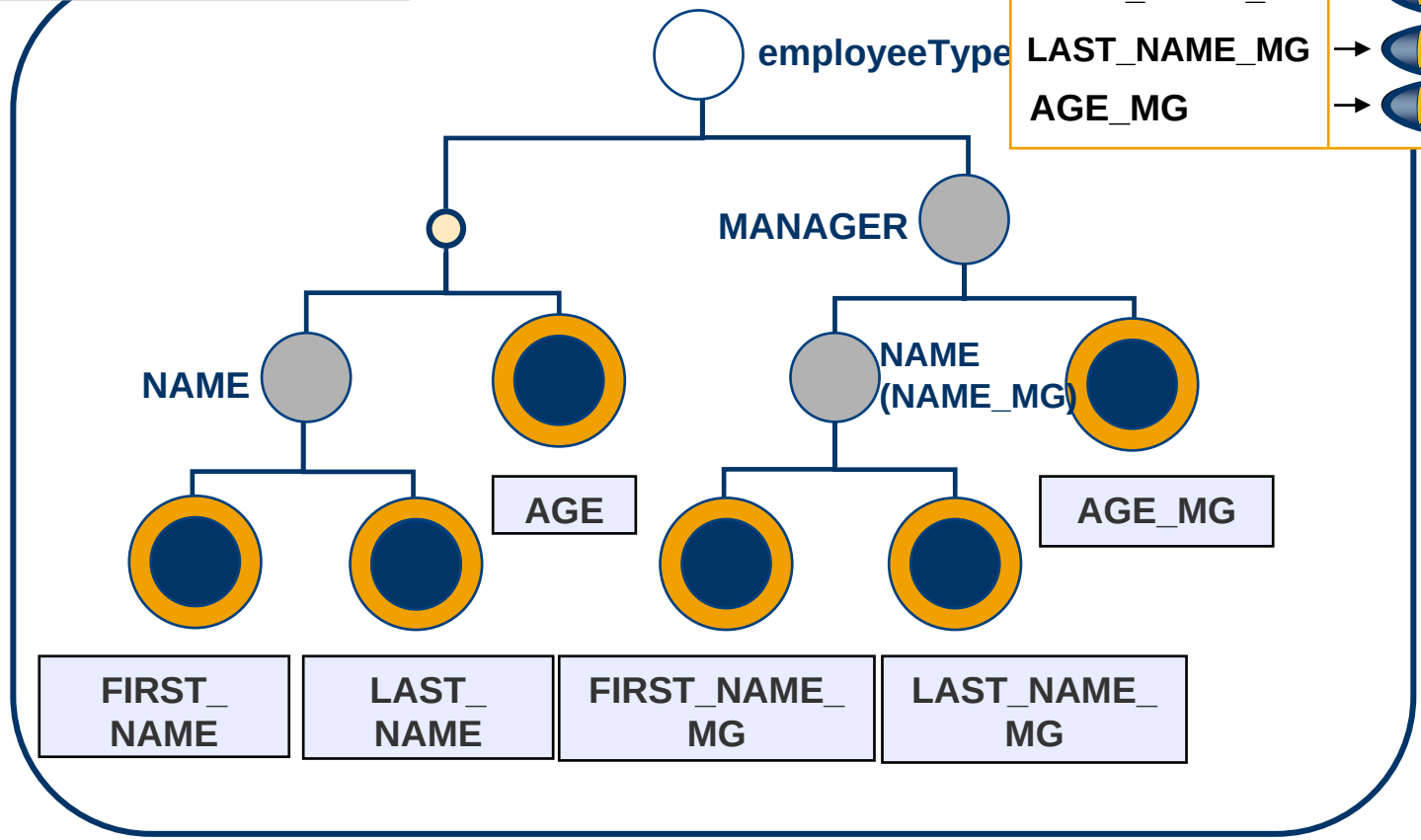


Included Views with Level Specification

```
TYPES:
  BEGIN OF nameType,
    first(30) TYPE c,
    last(40)  TYPE c,
  END OF nameType.
```

get_included_view( ) or
get_includel_view(>=2)

NAME	TYPE
FIRST_NAME	c(30)
LAST_NAME	c(40)
AGE	n(3)
FIRST_NAME_MG	c(30)
LAST_NAME_MG	c(40)
AGE_MG	n(3)

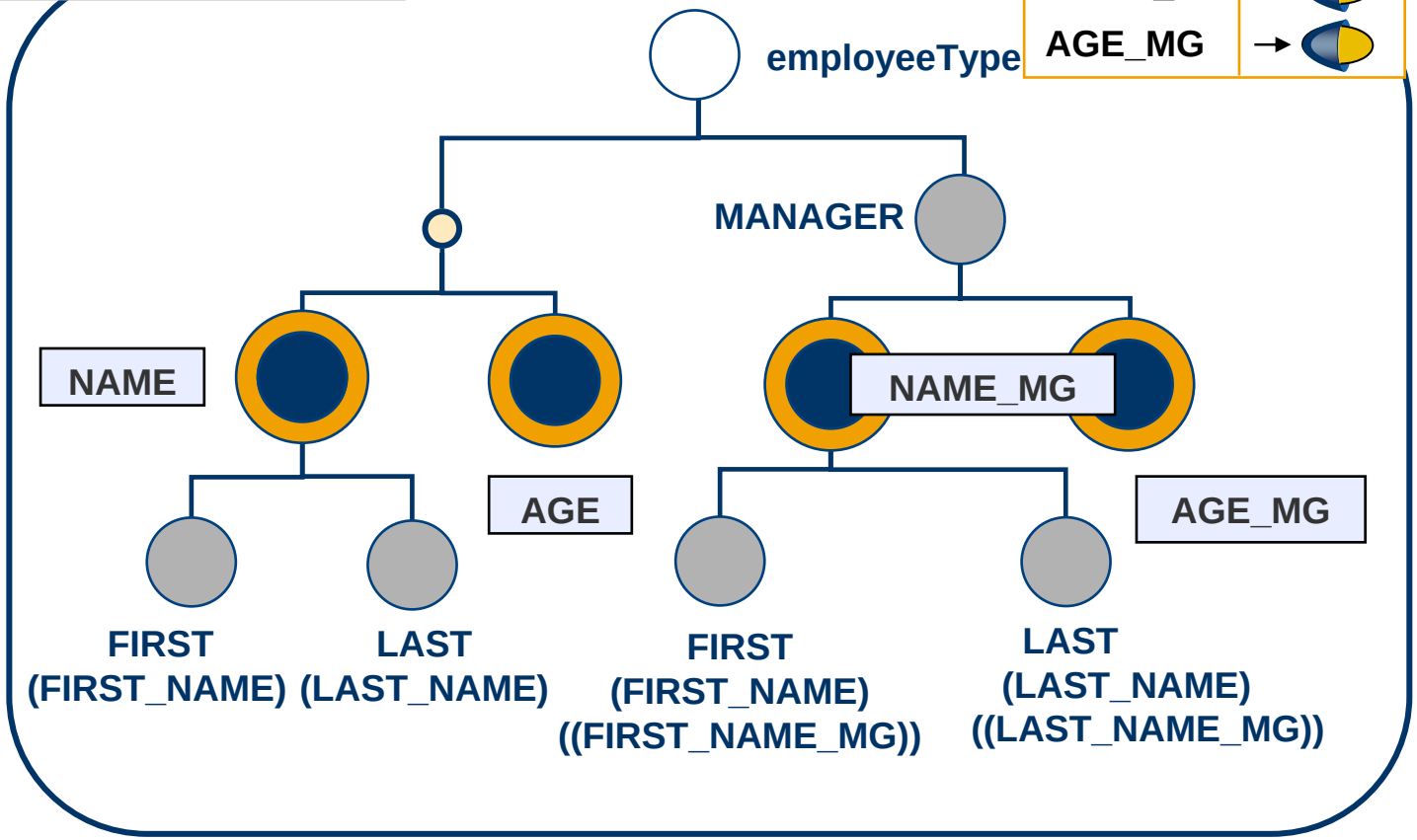


Included Views with Level Specification

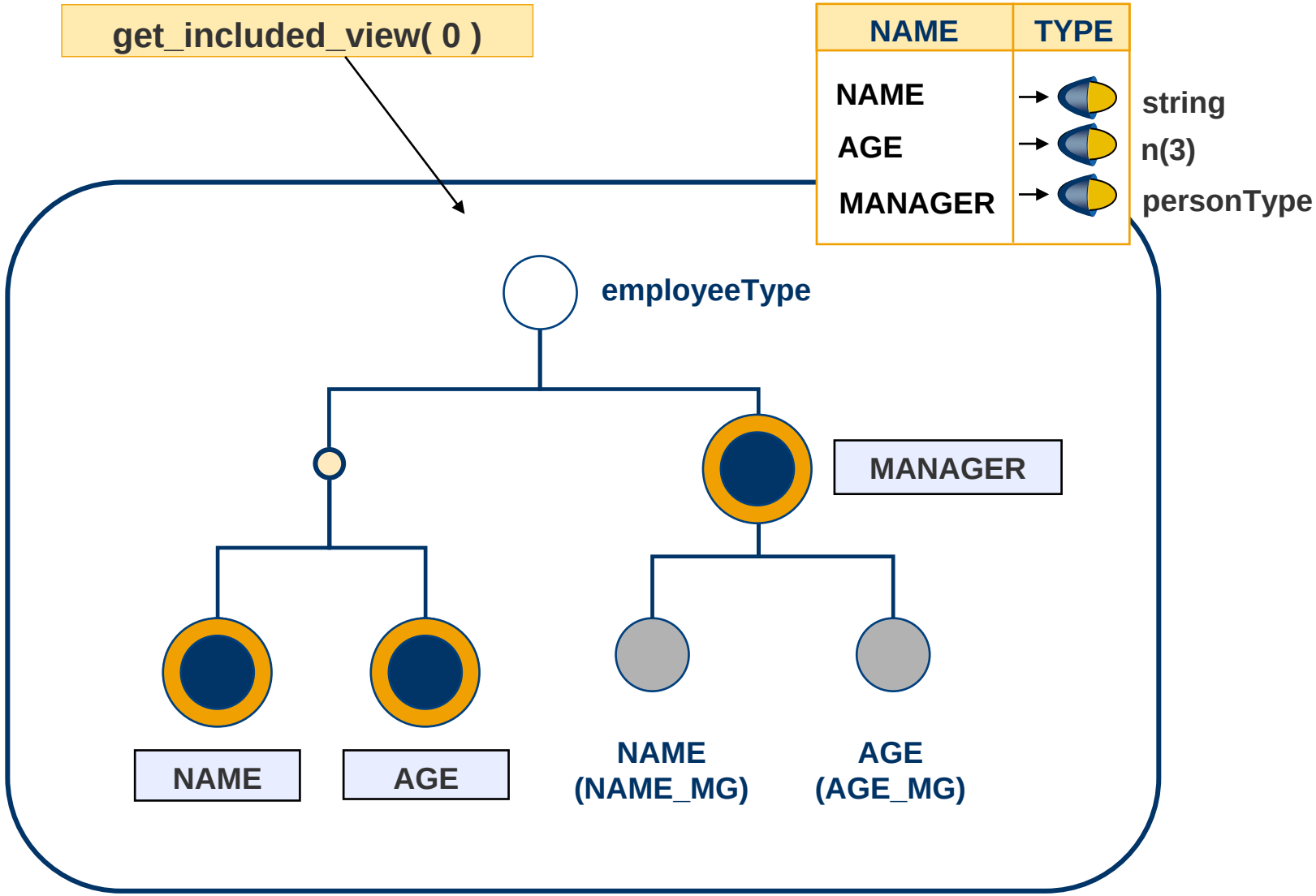
```
TYPES:
  BEGIN OF nameType,
    first(30) TYPE c,
    last(40)  TYPE c,
  END OF nameType.
```

get_included_view(1)

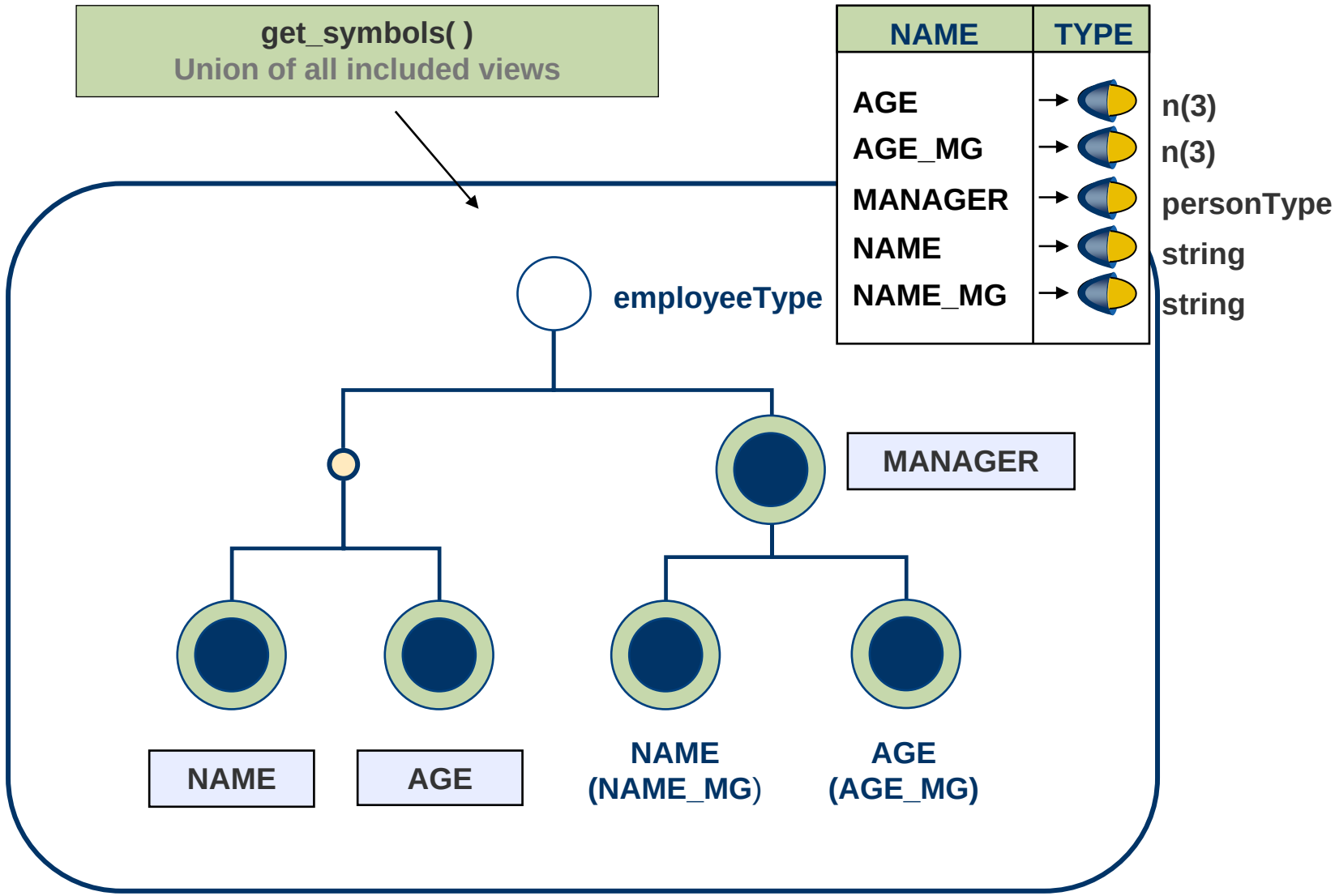
NAME	TYPE
NAME	nameType
AGE	n(3)
NAME_MG	nameType
AGE_MG	n(3)



Level-0-Views



Symbol Table



Summary Structure Analysis

Aims of Included Views and Symbol Table

- Get rid of include-specific part of the component description table
- Determine correct component selector (ASSIGN COMPONENT (comp) OF STRUCTURE ...)
- Easy loop over all components

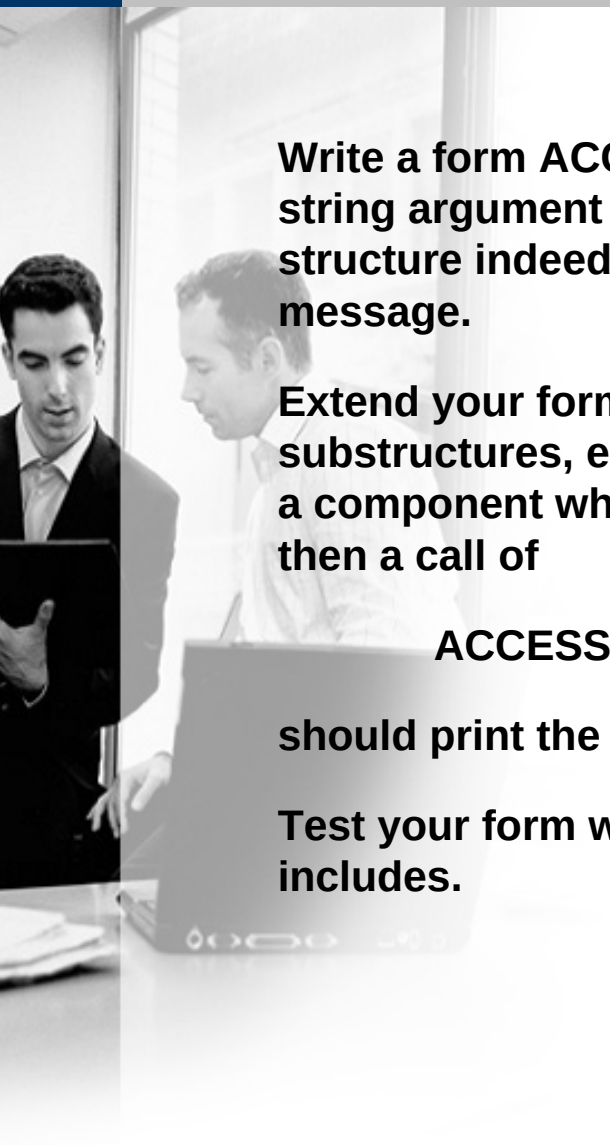
What are Included Views good for

- Easy access to all components of a structure without the hassle of empty component names (due to unnamed includes) and suffix computations
- Level-0-View: projection to first 2 columns of component description table with *unnamed includes* resolved
- View without Level Spec: projection to first 2 columns of component description table with *all nested includes* resolved (leaf view)
- Level-<l>-View: projection to first 2 columns of component description table with *nested includes resolved up to level <l>*.

What is the Symbol Table good for

- Easy answer to question: "is <name> a valid component name"

Exercise: Generic component access



Write a form ACCESS_COMPONENT which accepts any structure and a string argument that denotes a component of this structure. If the structure indeed has this component, write its value, else write an error message.

Extend your form in a way that it can also access subcomponents of substructures, e.g. if you pass structure 'foo' that has structure 'bar' as a component which again is a structure having 'toto' as a component, then a call of

ACCESS_COMPONENT (foo, 'bar-toto')

should print the corresponding value.

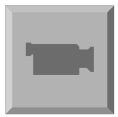
Test your form with some nested structures, preferable with additional includes.

Exercise: Understanding Included Views

In order to get a better understanding of Included Views and the Symbol Table write forms that dump

- **the component Table of a structure**
- **recursively also nested component tables**
- **the Included Views of the structure**
- **the symbol table of the structure**

Test your forms with some complicated structures.



Demo

Summary

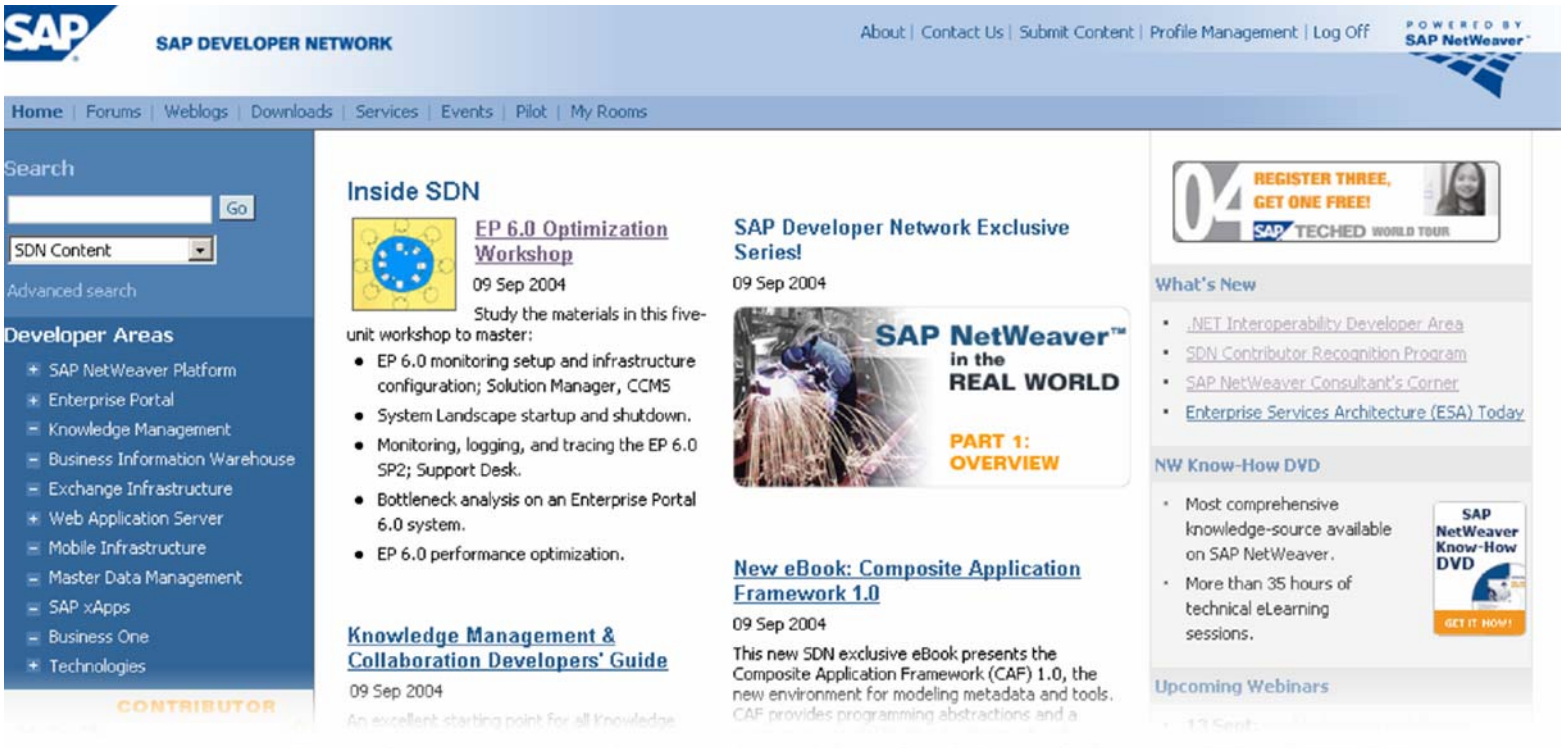
You are now able to:

- **Use generic types to make programs more flexible**
- **Write generic services which can work on arbitrarily structured data**
- **Explain the different kinds of genericity we have in ABAP**
- **Understand the RTTS**

Look for SAP TechEd ’04 presentations and videos on the SAP Developer Network.

Coming in December.

http://www.sdn.sap.com/



Questions?

Q & A

Feedback

Please complete your session evaluation.

**Be courteous — deposit your trash,
and do not take the handouts for the following session.**

Thank You !

Copyright 2004 SAP AG. All Rights Reserved