From the textbook—Chapter Three

Role of NetWeaver in ABAP development

3 layered architecture:

**Presentation Layer**
- Web Dynpro ABAP—web-based interfaces
- Business Server Pages (BSP)—server-side ABAP for the Web
- Dynpro—SAP GUI interfaces

**Application Layer**
- ABAP Objects
- XSLT and ST—ABAP ↔ XML
- JavaScript—implemented as a server-side script to support BSPS

**Persistence Layer**
- Open SQL—translates standardized SQL into DBMS-specific SQL
- Native SQL—non-translated DBMS-specific SQL
### From the textbook—Chapter Three

**ABAP program design**
- **Global declaration part**
  - Creation of data types, data objects, objects, and other structures
- **Implementation part**
  - Code logic using items from global declaration

**Launching ABAP programs using SA38**

**Launching ABAP programs via Transaction Code**
- Dialog transactions—designated program loaded and run
- OO transactions—designated method of selected object is run

**Launching ABAP programs programmatically**
- CALL TRANSACTION programe, LEAVE TO TRANSACTION programe

### From the textbook—Chapter Three

**ABAP program types**
- **Class pools**—global class definition. Created via Class Builder. No typical source code.
- **Interface pools**—global interface definition. Created via Class Builder.
- **Function groups/pools**—function libraries
- **Subroutine pools**
- **Executable programs**
- **Module pools**
- **Type groups**—defines types for general global use

**Includes**
- Source code include via INCLUDE programe
- Top Include—global declarations visible to included programs
Discussion Topics

- Analyzing Runtime Errors with ST22
- Defining and Calling Subroutines
  - Parameter Passing
- Messages
- String Manipulation in ABAP
- Complex Data Types
  - Structures
  - Internal Tables
  - Internal Table Operations

Analyzing Runtime Errors

In the event a runtime error causes a program to crash, transaction ST22 is used to analyze the dump produced by the crash.
Subroutines

Similar to functions in other languages. Provides encapsulation of functionality. Permits parameter passing using different methods.

Basic Syntax:

```
FORM mysubroutine. "subroutine definition
  1 or more statements here.
ENDFORM.

PERFORM mysubroutine. "subroutine call
```

It is typical for subroutine definitions to be placed at the end of program source code. (internal subroutine)

Subroutines can see and manipulate global program data, but that is considered very poor design. (Violates principle of encapsulation.)
Parameter passing to subroutines

Actual parameters (caller) are copied/transferred to formal parameters listed in the subroutine (callee) interface.

Parameter passing methods:

- **Call by Value**—formal parameter value copied to actual parameter.
- **Call by Reference**—actual parameter becomes local alias for formal parameter. Subroutine operates on original formal parameter (no copy made). Changes to actual parameter effective in formal parameter when subroutine terminates.
- **Call by Value and Result**—formal parameter value copied to actual parameter. At end of subroutine (if reached successfully), the actual parameter's final value is copied back to the formal parameter.

Parameter typing

Actual parameters are typed upon data object creation in the main program.

Formal parameters may be listed without type or with TYPE ANY. This **generic typing** allows parameters of any type to be passed to the subroutine.

Generic typing may result in a runtime error if an operation is performed which is invalid for the type of data received.

Formal parameters may be listed with a TYPE clause to set a fixed type. This **concrete typing** may result in an automatic type conversion attempt (if possible).

For incomplete data types (p, n, c, x) a LENGTH clause is not set. Formal parameters may use user-defined local or global types.
Parameter passing mechanics—call by value

PERFORM mysubroutine "subroutine call
USING
   mainvar1.

FORM mysubroutine "subroutine definition
USING
   value(subvar1) TYPE i.
   1 or more statements here.
ENDFORM.

Note subroutine definition places formal parameter name within value(). This indicates it is the value of the data object which is passed (i.e. it is copied).

The call lists the actual parameter (note that value() is not indicated).

Use of USING indicates that the value of the actual parameter in the caller cannot change based on the subroutine.

Parameter passing mechanics—call by reference

PERFORM mysubroutine "subroutine call
USING [or CHANGING]
   mainvar1.

FORM mysubroutine "subroutine definition
USING [or CHANGING]
   subvar1 TYPE c.
   1 or more statements here.
ENDFORM.

Note value() not used. This indicates that value is not copied. Subroutine is operating on the 'original' data object.

Since the subroutine is operating on the original actual parameter, changes it makes are effective on the original when the subroutine completes.

If the module will be making changes, use CHANGING. If it will not be making changes, use USING. The difference between USING and CHANGING in this context is strictly for clarity of design.
Parameter passing mechanics—call by value and result

PERFORM mysubroutine "subroutine call
   CHANGING
       mainvar1.
FORM mysubroutine "subroutine definition
   CHANGING
       value(subvar1) TYPE ANY.
   1 or more statements here.
ENDFORM.

The use of value() indicates that a copy of the data object's value is passed to the subroutine.
The use of CHANGING indicates that at the successful end of the subroutine, the final value of the formal parameter will be copied back to the actual parameter (caller).

Parameter Passing Syntax Summary

<table>
<thead>
<tr>
<th>Caller (actual parameters)</th>
<th>Callee (formal parameters)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Using</td>
</tr>
<tr>
<td>Call by value</td>
<td>✅</td>
</tr>
<tr>
<td>Call by value and result</td>
<td></td>
</tr>
<tr>
<td>Call by reference with no change in subroutine</td>
<td>✅</td>
</tr>
<tr>
<td>Call by reference with change in subroutine</td>
<td></td>
</tr>
</tbody>
</table>

Any time a copy operation is a part of parameter passing, the actual parameter will be listed within value() in the subroutine definition.

Any time the subroutine will change the value of the formal parameter, CHANGING is used with formal and actual parameters.

With Call by Reference use of USING is a design 'contract' to not change the value. This is not enforced by ABAP.
Test your Understanding

PERFORM mysubroutine
  USING
    mainvar1
    mainvar2
  CHANGING
    mainvar3
    mainvar4.

Identify the type of parameter passing used for each parameter shown.

FORM mysubroutine
  USING
    subvar1
    value(subvar2)
  CHANGING
    value(subvar3)
    subvar4.
ENDFORM.

Note the placement of periods in the syntax!

Parameter Passing—complete example

PERFORM mysubroutine
  USING
    mainvar1
    mainvar2
  CHANGING
    mainvar3
    mainvar4.

Note the placement of periods in the syntax!

FORM mysubroutine
  USING
    subvar1
    value(subvar2)
  CHANGING
    value(subvar3)
    subvar4.
ENDFORM.
## Parameter passing rules

Although a subroutine can operate on global data, this is poor design and should never be done. The risk of maintenance errors is greatly increased.

Select the most conservative method of parameter passing based on design goals:

- **Call by value** has the benefit of providing data object security. Changes in the variable in the subroutine will not affect the main data object.

- Call by value and call by value and result have the disadvantage of overhead associated with making data object copies. Avoid for large data objects.

## Quick Practice
Exiting subroutines

A subroutine terminates normally if the flow of control reaches **ENDFORM** or a **RETURN** statement is reached. Flow of control is returned to the statement after the calling statement.

Call by Value and Return parameters written back to calling parameters before returning flow of control.

A **STOP** statement or display of an **error message** causes the subroutine to terminate and the overall program to exit.

Call by Value and Return parameters **not** written back to calling parameters.

Sending Messages

```
MESSAGE 'message text' TYPE '[type]'.
```

Used to note special program conditions. Types of messages:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Behavior</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>information</td>
<td>Display message. Program continues after user dismisses message.</td>
<td>Modal dialog box</td>
</tr>
<tr>
<td>S</td>
<td>set message</td>
<td>Display message. Program continues without interruption</td>
<td>Status bar or modal dialog box*</td>
</tr>
<tr>
<td>W</td>
<td>warning</td>
<td>Display message. Behavior depends on context.</td>
<td>Status bar or modal dialog box*</td>
</tr>
<tr>
<td>E</td>
<td>error</td>
<td>Display message. Behavior depends on context.</td>
<td>Status bar or modal dialog box*</td>
</tr>
<tr>
<td>A</td>
<td>termination</td>
<td>Program terminated</td>
<td>Modal dialog box</td>
</tr>
<tr>
<td>X</td>
<td>short dump</td>
<td>Runtime error triggered with dump</td>
<td>Dump to screen</td>
</tr>
</tbody>
</table>

*Depends on GUI settings
Useful ABAP Statements: String Manipulation

ABAP provides a number of language constructs for working with string-type data. These work for data types string, c, d, t, and n.

CONCATENATE s1 s2 INTO s3 [SEPARATED BY s4].

S3 must be of sufficient size or the new string will be truncated.
String Manipulation

FIND s1 IN s2.
FIND [FIRST OCCURRENCE OF or ALL OCCURRENCES OF] s1 IN s2.
FIND s1 IN s2 [IGNORING CASE or RESPECTING CASE].
  Result of find operation stored in sy-subrc.
  sy-subrc set to 0 if found.
  sy-subrc set to 4 if not found.
FIND ALL OCCURRENCES OF s1 IN s2 MATCH COUNT x.
  Number of found strings stored in x.
Regular expression processing is possible as well. (FIND REGEX...)

String Manipulation

REPLACE s1 IN s2 WITH s3.
REPLACE [FIRST OCCURRENCE OF or ALL OCCURRENCES OF] s1 IN s2 WITH s3.
REPLACE s1 IN s2 [IGNORING CASE or RESPECTING CASE] WITH s3.
  Result of replace operation stored in sy-subrc.
  sy-subrc set to 0 if a replacement was done.
  sy-subrc set to 2 if a replacement was done, but the result was truncated due to variable size.
  sy-subrc set to 4 if not found.
REPLACE ALL OCCURRENCES OF s1 IN s2 WITH s3 REPLACEMENT COUNT x.
  Number of replaced strings stored in x.
String Manipulation

\texttt{SPLIT s1 AT char INTO s2 s3 [s4 ...].}

Can create as many substrings as relevant.

If insufficient substrings listed, the remaining text and the separator are placed into the last substring listed.

Quick Practice
Complex data types--structures

Complex data types can be defined to create structured variables (also called structures)

**TYPES:**

```
BEGIN OF struct_type_name,
  field_name TYPE type,
  field_name TYPE type, [... as needed
END OF struct_type_name.
```

Components of a structure are referenced using a hyphen:

```
structure_variable_name – structure_component_name
```

Structures can be nested.

Structure variable a1 can be copied to structure variable a2 using:

```
MOVE-CORRESPONDING a1 TO a2.
```

Only like name components are copied. All others ignored.

---

Structure Example

**TYPES:**

```
BEGIN OF address_struct,
  address1 TYPE c LENGTH 60,
  address2 TYPE c LENGTH 60,
  city TYPE c LENGTH 60,
  state TYPE c LENGTH 2,
  zip TYPE n LENGTH 5,
  zipext TYPE n LENGTH 4,
END OF address_struct.
```

```
DATA str TYPE address_struct.
str-address1 = '10061 Bristol Park Road'.
str-city = 'Cantonment'.
str-state = 'FL'.
str-zip = '37614'.
str-zipext = '6610'.
```
Practice Working with Structures

Introducing SAP's training database tables

SAP provides a set of database tables based on airplane flight related data (flights, airports, meals, customers, etc.)

Some tables of interest:
- SAIRPORT – airport information
- SCARR – airline information
- SPFLI – flight schedules
- SBOOK – flight bookings
- SCUSTOM – flight customers

Transaction SE16 can be used to view table contents
Complex data types—Internal Tables

Similar to database table concept, but stored in memory. Number of records restricted only by capacity of system. Useful for temporarily storing data from database for processing, preparing data for output, or preparing data for other use. Data in an internal table can be accessed by searching the table based on key column values or indexes (row numbers).

Keys may be unique or non-unique. Searching an internal table with non-unique keys may result in more than 1 record returned per search.

Three types of internal tables:
- **Standard**—similar to DB table (row and key access possible)
- **Sorted**—data always sorted by keys (row and key access possible)
- **Hashed Table**—keys must be unique (only key access possible)

Internal Table TYPE Creation

A internal table is a dynamic sequence of items of the same type—typically structures, but other data types are possible.

**TYPES:**
- `Table_type_name` TYPE [table type] TABLE OF [data type]

**TYPES:**
- `address_table_type` TYPE STANDARD TABLE OF address_struct
- WITH NON-UNIQUE KEY address1 zip.

A standard table can only have non-unique keys, hashed tables can only have unique keys, and sorted can have either.
**Internal Table Creation and Data Filling Example**

Training table SCARR is of type SBC400_T_SCARR.

Use SE11 to view the table and data type in ABAP Dictionary.

Create an internal table of the same type as SCARR:

```
DATA itable TYPE sbc400_t_scarr.
```

Create a structure that matches the columns contained in the internal table just created:

```
DATA str LIKE LINE OF itable.
```

Copy the contents of database table to an internal table:

```
SELECT * FROM scarr INTO TABLE itable.
```

Write the contents of the internal table to the screen as a list

```
LOOP AT itable INTO str.
  WRITE: / str-mandt, / str-carrid, / str-carrname, 
          / str-currcode, / str-url, /.
ENDLOOP.
```

---

**Single Record Table Operations**

**Append**

Adds data object to the end of the table

```
APPEND data_obj TO table_obj.
```

**Insert**

In standard table, same as append. In sorted table, inserted based on sort order. In hashed table, inserted based on hash algorithm.

```
INSERT data_obj INTO TABLE table_obj.
```

**Read**

Copies a table row into a specified data object

Row selection can be based on index number or table key.

```
READ TABLE table_obj INDEX n INTO data_obj.
READ TABLE table_obj WITH TABLE KEY field = 'value' INTO data_obj.
```
## Single Record Table Operations

### Change
Changes a table row

- Based on index:
  ```
  MODIFY tbl_obj INDEX n FROM obj.
  MODIFY tbl_obj INDEX n FROM obj TRANSPORTING comp [comp].
  ```

- Based on key value:
  ```
  MODIFY TABLE tbl_obj FROM obj TRANSPORTING comp [comp].
  ```

### Delete
Deletes a row

- ```
  DELETE tbl_obj INDEX n.
  DELETE TABLE tbl_obj WITH TABLE KEY key = value.
  ```

## Multiple Record Table Operations

### LOOP AT...ENDLOOP
Loops through entire table (or portion) copying rows into a data object.

- ```
  LOOP AT tbl_obj INTO obj.
  LOOP AT tbl_obj INTO obj FROM index.
  LOOP AT tbl_obj INTO obj TO index.
  LOOP AT tbl_obj INTO obj WHERE col = val.
  ```

### DELETE
Deletes row(s) matching condition

- ```
  DELETE tbl_obj FROM index.
  DELETE tbl_obj TO index.
  DELETE tbl_obj WHERE col = val.
  ```
Multiple Record Table Operations

**INSERT**
Copies rows from one internal table to another

```
INSERT LINES OF tbl_obj [condition] INTO tbl2_obj [condition].
```

**Append**
Appends rows from one internal table to another

```
APPENDS LINES OF tbl_obj [condition] TO tbl2_obj.
```

Quick Practice