Object Services in ABAP™
Contents

Preface .......................................................................................................................... 9

1 Introduction .............................................................................................................. 11

2 Reading Persistent Objects ...................................................................................... 19

  2.1 Creating a Persistent Class .................................................................................. 20
  2.1.1 Selecting a Database Table for Persistence Representation ................................................................. 21
  2.1.2 Assigning the Individual Fields to the Attributes of the Class ................................................................. 22
  2.1.3 Completeness of the Persistence Representation ......................................................... 26
  2.1.4 Instance GUIDs and Business Keys ....................................................................... 26
  2.1.5 Multiple-Table Mapping ................................................................................... 28
  2.1.6 Transient Attributes ..................................................................................... 29
  2.1.7 Activating the Persistent Class ........................................................................ 30

  2.2 Instantiating Persistent Objects .......................................................................... 30
  2.2.1 IF OS CA PERSISTENCY~GET_PERSISTENT_BY_OID ................................................. 31
  2.2.2 IF OS CA PERSISTENCY~GET_PERSISTENT_BY_KEY ......................................... 33
  2.2.3 GET_PERSISTENT ............................................................................................. 34
  2.2.4 Exception Handling .......................................................................................... 35
  2.2.5 Behavior for Already-Loaded Objects .................................................................. 36

  2.3 Reading Attribute Values ..................................................................................... 37

  2.4 Persistent References .......................................................................................... 38
  2.4.1 Definition of a Persistent Reference in the Persistence Representation ................................................................. 40
  2.4.2 Runtime Behavior ........................................................................................... 41
  2.4.3 Exception Handling .......................................................................................... 42
  2.4.4 Limitations for the Use of Persistent References .............................................. 43

  2.5 Inheritance .......................................................................................................... 43
  2.5.1 Vertical Mapping .............................................................................................. 44
  2.5.2 Horizontal Mapping ........................................................................................ 45
  2.5.3 Comparing the Two Mapping Variants .............................................................. 47
  2.5.4 Type Identifier .............................................................................................. 47
3 Creating and Changing Persistent Objects

3.1 Creating Persistent Objects

3.1.1 IF_OS_FACTORY~CREATE_PERSISTENT

3.1.2 IF_OS_FACTORY~CREATE_PERSISTENT_BY_KEY

3.1.3 CREATE_PERSISTENT

3.1.4 Exception Handling

3.1.5 Creating Transient Objects

3.2 Changing Attribute Values

3.3 Transaction Service

3.3.1 Transactions in SAP Systems

3.3.2 Object-Oriented Transactions

3.3.3 Concatenating Transactions

3.3.4 Subtransactions

3.3.5 Transaction Modes

3.3.6 Update Modes

3.3.7 Check Agents

3.3.8 Undo Mechanism for Persistent Objects

3.3.9 Transaction Statuses

3.4 Management States of Persistent Objects

3.4.1 State Transitions for Persistent Objects That Don’t Exist in the Database Yet

3.4.2 State Transitions for Persistent Objects That Already Exist in the Database

3.4.3 State Transitions When Finishing Transactions and SAP Logical Units of Work

3.5 Summary

4 Selecting Persistent Objects

4.1 Determining Keys of Persistent Objects with Open SQL

4.2 Mass Instantiation

4.3 Simple Selections Using the Query Service

4.4 More Complex Selections Using the Query Service

4.4.1 Defining Query Parameters
## Contents

4.4.2 Defining the Filter Condition ............................................ 99  
4.4.3 Defining the Sort Condition .............................................. 106  
4.4.4 Passing Concrete Values for Query Parameters ................. 109  
4.5 Comparing the Query Service and Open SQL ....................... 112  
4.6 Handling Newly Created and Changed Objects ........................ 113  
4.7 Summary ............................................................................. 116  

5 Internal Structure and Functioning of Object Services .......... 117  
5.1 Persistence Service .......................................................... 117  
5.1.1 Class Agent and Base Agent ........................................... 117  
5.1.2 Instance Manager and Persistency Manager ...................... 121  
5.1.3 Database Tables with Information on the Persistence  
   Representation ..................................................................... 124  
5.1.4 Automatic Source Code Generation ................................. 124  
5.1.5 Garbage Collection in ABAP Objects ............................... 127  
5.2 Transaction Service .......................................................... 129  
5.2.1 Implementation of the Transaction Manager and the  
   Transactions ..................................................................... 129  
5.2.2 Undo Management ......................................................... 131  
5.3 CL_OS_SYSTEM .................................................................. 134  
5.4 Summary ............................................................................. 136  

6 Useful Enhancements for Practical Use ................................. 137  
6.1 Reloading Objects from the Database ................................. 137  
6.2 Release of Objects No Longer Required ............................... 140  
6.3 Conversion Between Object and Structure ......................... 141  
   6.3.1 Reading Attribute Values from a Persistent Object .......... 142  
   6.3.2 Writing Attribute Values to a Persistent Object .......... 144  
   6.3.3 Structures in Connection with Persistent Classes .......... 146  
6.4 Use of Persistent Objects in User Interfaces ......................... 150  
   6.4.1 SAP Control Framework ................................................. 150  
   6.4.2 Using Persistent Objects in Web Dynpro Contexts .......... 154  
   6.4.3 Setting the Transaction Mode in a  
      Web Dynpro Application ............................................... 158  
6.5 Summary ............................................................................. 159
## Contents

### 7 Intelligent Persistent Objects .................................................. 161

- **7.1 Plausibility Checks** ............................................................... 161
  - **7.1.1 Variant 1: Custom Access Methods with Different Names** in the Persistent Class ....................................................... 163
  - **7.1.2 Variant 2: Delegation with Access Methods in a** Usual Class ................................................................. 165
  - **7.1.3 Variant 3: Persistent Subclass** ........................................ 167
  - **7.1.4 Variant 4: Enhancement Framework** ............................. 169
  - **7.1.5 Comparing the Various Variants** .................................. 174
- **7.2 Lazy Loading** .................................................................. 176
- **7.3 Summary** ......................................................................... 180

### 8 Integration of the SAP Lock Concept and Object Services ...... 181

- **8.1 The SAP Lock Concept** ...................................................... 182
- **8.2 Pessimistic and Optimistic Locking** .................................... 186
  - **8.2.1 Lock Mode** ................................................................. 186
  - **8.2.2 Pessimistic Locking** .................................................. 187
  - **8.2.3 Optimistic Locking** .................................................. 189
  - **8.2.4 Using the Two Locking Strategies in an SAP System** ..... 191
- **8.3 Integration of Optimistic Locking** ....................................... 192
  - **8.3.1 Setting Optimistic Locks** .......................................... 193
  - **8.3.2 Registering the Check Agent** ...................................... 197
  - **8.3.3 Converting Optimistic Locks Into Exclusive Locks** ....... 199
- **8.4 Integration of Pessimistic Locking** ..................................... 204
- **8.5 Integration of Both Locking Strategies** ............................... 205
- **8.6 Summary** ........................................................................ 207

### 9 Conclusion ............................................................................ 209

- The Authors ............................................................................. 211
- Index .......................................................................................... 213
Various options are available for identifying persistent objects that meet certain criteria. This chapter introduces these options, including classic Open SQL and the Object Services Query Service.

4 Selecting Persistent Objects

The previous chapters described how you can instantiate objects whose keys — that is, either their business keys or their instance GUIDs — you already know. In real life, however, you often have to instantiate objects whose keys you don’t know. This chapter explains how to instantiate all objects that meet certain criteria. Criteria can be, for example, a part of a composed key, specific values or value ranges of attributes, or the class to which the objects belong.

4.1 Determining Keys of Persistent Objects with Open SQL

Open SQL statements, which are integrated in ABAP and can be used independently of the database system, enable you to determine the keys of the objects that you want to instantiate. You can thus instantiate all objects that meet specific criteria without having to use other components of Object Services than the ones already introduced.

Listing 4.1 follows this approach. A SELECT query in Open SQL first determines the business keys of all flights from the SAP flight data model that take place at the current date and temporarily stores them in an internal table. In a LOOP loop for the internal table of the business keys, the class agent then instantiates the corresponding persistent object in each loop run for a business key using the IF_OSS_CA_PERSISTENCY~GET_PERSISTENT_BY_KEY method. An additional internal table finally stores the references to the instantiated persistent objects.

```
DATA:
  rf_ca_sflight TYPE REF TO /iot/ca_sflight,
  rf_flight TYPE REF TO /iot/cl_sflight.
```
ta_business_keys_flights TYPE STANDARD TABLE OF 
scol_flight_key,
ta_rf_flights TYPE STANDARD TABLE OF 
REF TO /iot/cl_sflight,
w_business_key_flight TYPE scol_flights_key.

rf_ca_sflight = /iot/ca_sflight=>agent.

* Determine business keys of all flights with today's date
* with an Open SQL query
SELECT carrid connid fldre INTO CORRESPONDING FIELDS OF TABLE 
ta_business_keys_flights 
FROM sflight 
WHERE fldre = sy-datum.

* Instantiate for each business key the corresponding
* persistent object separately
LOOP AT ta_business_keys_flights INTO wa_business_key_flight.
    rf_flight ?= 
        rf_ca_sflight->if_os_ca_persistency~get_persistent_by_key(
            wa_business_key_flight).
    APPEND rf_flight TO ta_rf_flights.
ENDLOOP.

Listing 4.1 Instantiating All Flights on the Current Date with Open SQL and the GET_PERSISTENT_BY_KEY Method

Before Release 7.0 of SAP NetWeaver AS ABAP, the combination of an Open SQL statement and an instantiation that is implemented for each object separately was the only option to instantiate all persistent objects that meet specific criteria. As of Release 7.0, SAP NetWeaver AS ABAP delivers two new alternatives for instantiating multiple persistent objects.

Since then, the class agents of persistent classes have provided two new mass instantiation methods. These methods enable you to use one method call to instantiate the persistent objects for multiple already known keys.

Even more comprehensive are the innovations that a third newly introduced method in the class agent provides. You pass an object that contains conditions to this method. This object is called a query. The method then instantiates all persistent objects that meet the conditions of the query. You don't have to use Open
SQL statements in your application in this case. The functions for formulating and executing queries are referred to as the Query Service. The Persistence Service, the Transaction Service, and the Query Service constitute the three current Object Services components.

4.2 Mass Instantiation

To instantiate persistent objects for multiple already known keys, every class agent contains a method to which you can pass an internal table with business keys or instance GUIDs. Like Listing 4.1, Listing 4.2 instantiates all flights on the current date. The business keys of the persistent objects again are determined by an Open SQL statement. Then, the GET_PERSISTENT_BY_KEY_TAB method from the IF_OS_CA_PERSISTENCY interface is called. This method tries to load the appropriate persistent object from the database for every transferred business key. The method returns an internal table with references to the instantiated persistent objects. The reference to a persistent object is respectively specified in the result table in the same line that contained the appropriate key in the transferred table.

```plaintext
DATA: 
  rf_ca_sflight TYPE REF TO /iot/ca_sflight,
  ta_business_keys_flights TYPE STANDARD TABLE OF 
    scol_flights_key,
  ta_ro_flights TYPE osreftab.

rf_ca_sflight = /iot/ca_sflight=>agent.

* Determine business keys of all flights with today's date
* with an Open SQL query
SELECT carrid connid fldate 
  INTO CORRESPONDING FIELDS OF TABLE 
    ta_business_keys_flights 
  FROM sflight 
  WHERE fldate = sy-datum.

* Instantiate for each business key the corresponding
* persistent object
  ta_ro_flights = 
    rf_ca_sflight->if_os_ca_persistency~get_persistent_by_key_tab( 
      ta_business_keys_flights ).

Listing 4.2 Instantiating All Flights on the Current Date with Open SQL and the GET_PERSISTENT_BY_KEY_TAB Method
```
The `GET_PERSISTENT_BY_KEY_TAB` method is available in the class agents of all persistent classes for which you have defined a business key in the persistence representation. The transferred internal table can be a standard table or a sorted table. The line type of the table must correspond to the business key structure of the persistent class. In this context, remember that you have to use a structure in which the individual components are sorted by attribute names (see the warning box in Section 2.2.2, IF.OS_CA_PERSISTENCY~GET_PERSISTENT_BY_KEY, of Chapter 2).

The return value of the method is of the `OSREFTAB` type. This is a standard table in which each line contains a reference of the `OBJECT` type. In case of later accesses to the persistent objects, downcasting is required here.

Similar to the `GET_PERSISTENT_BY_KEY_TAB` method, the class agent contains the `GET_PERSISTENT_BY_OID_TAB` method, which enables you to instantiate multiple persistent objects via their respective instance GUID. You can use the method in the class agent of persistent classes in whose persistence representation an instance GUID is defined. You pass a standard table or a sorted table with the `OS_GUID` line type to the method and also receive a result of the `OSREFTAB` type by this method.

The extent to which you can improve the speed, which can be achieved by implementing mass instantiation instead of calling the instantiation of one persistent object respectively several times (see Listing 4.1), depends on the system and the type of the objects used. Considerable speed improvements of more than 50% as well as a runtime behavior that hardly differs in both variants have been observed. As of Support Package 13 for SAP NetWeaver AS ABAP 7.0, however, the process speed of the mass instantiation has in general always been as high as the speed of the separately called instantiation process. In previous releases (due to an error that has been eliminated in the meantime), it was possible that the mass instantiation process, if more than 500 persistent objects were instantiated in parallel, ran slower than the instantiation process that is subsequently called for individual objects several times.

The methods for the mass instantiation only raise an exception of the `CX_OS_OBJECT_NOT_FOUND` class if they detect an object that is in the `DELETED` or `TRANSIENT` management state for a transferred key. In an attribute, the exception object contains the business key or instance GUID of the object for which the exception
was raised. If such an exception occurs, the corresponding method terminates the instantiation of further persistent objects for the transferred keys.

If you pass a key for which no object exists in the database to a method for mass instantiation, this doesn’t raise an exception as it’s the case for an instantiation of individual objects. The instantiation of further objects is also not terminated in this case. Instead, the result table contains a null reference in the corresponding line. For example, if you pass a key to the method in the third line of the table, and the method can’t find a persistent object for this key in the class agent management or in the database, the result table contains a null reference and not a reference to a persistent object in the third line. Each line of the result table for which you have passed a key of an existing persistent object contains a valid reference.

4.3 Simple Selections Using the Query Service

Besides mass instantiation that enables you to instantiate multiple persistent objects for already known keys, Object Services also provide the option for instantiating all persistent objects whose persistent attributes meet certain conditions. The Query Service helps you formulate a query in which you define a filter condition specifying the conditions that the persistent attributes of the persistent objects are supposed to meet. You can also define a sort condition of a query according to which criteria the system is supposed to sort the persistent objects.

There are two options for formulating the filter condition and the sort condition: You can use a syntax that is similar to the WHERE or ORDER BY clauses of an SQL query, and you can define filter and sort conditions using objects. This alternative way of formulating conditions is described in Section 4.4, More Complex Selections Using the Query Service.

Listing 4.3 provides an example of how you can implement the determination of all flights on the current date. Here, the flights are determined and instantiated using the Query Service.

```
DATA: rf_ca_sflight TYPE REF TO /iot/ca_sflight,
       ri_query_manager TYPE REF TO if_os_query_manager,
       ri_query TYPE REF TO if_os_query,
       ta_ro_flights TYPE osreftab,
       v_filter TYPE string.
```
rf_ca_sflight = /iot/ca_sflight=>agent.
ri_query_manager = cl_os_system=>get_query_manager( ).

* Formulate filter condition and create query
CONCATENATE 'FLDATE = ' sy-datum ' ' INTO v_filter.
ri_query =
  ri_query_manager->create_query(
    i_filter = v_filter ).

* Instantiate all persistent objects that correspond to the
  * conditions of the query
ta_ro_flights =
  rf_ca_sflight->if_os_ca_persistency~get_persistent_by_query(
    ri_query ).

Listing 4.3 Instantiating All Flights on the Current Date Using the Query Service

Creating a query of the Query Service is similar to creating a transaction of the
Transaction Service: The static method of the CL_OS_SYSTEM class (here: GET_QUERY_>
MANAGER) provides you with a reference to the Query Manager. The Query Manager
provides a method called CREATE_QUERY, which creates a new query and returns
a reference to this query. You can pass the filter conditions and sort conditions to
the CREATE_QUERY method when creating the query, or you can set them via vari-
ous methods of the query object after you have created the query (see Section 4.4,
More Complex Selections Using the Query Service).

When formulating the filter and sort conditions, you have to refer to the names
of the persistent attributes respectively. If you've assigned the same names to the
persistent attribute in the persistence representation as to the field in the database
table, this isn't a potential source of error. However, if you've defined attributes
names in the persistence representation that aren't identical to the field names in
the database table, you have to take extra care to use the attribute names instead
of the field names of the underlying database table when formulating the filter and
sort criteria. Also, in filter and sort conditions, you're only allowed to use persis-
tent attributes for which you have selected the Public visibility in the persistence
representation.

After defining the desired filter and sort conditions for the query, call the IF_OS_>
CA_PERSISTENCY~GET_PERSISTENT_BY_QUERY method of the class agent of the class
for which you want to instantiate the persistent objects. To this method, you must
pass the query object in which you defined the filter and sort conditions.
You can modify the behavior of the `IF_OS_CA_PERSISTENCY-GET_PERSISTENT_BY_QUERY` method using the `I_UPTO` and `I_SUBCLASSES` parameters:

- The `I_UPTO` parameter enables you to specify how many persistent objects the method is supposed to return as a maximum. This way, you can limit the number of instantiated objects and consequently the application’s load time and memory consumption. This is particularly useful if you allow the user to define the filter condition, and numerous persistent objects are available for the persistent class.

  If the number of persistent objects that meet the filter condition is greater than the value of `I_UPTO`, the method provides a subset of the persistent objects that meet the filter condition. If you’ve specified a sort condition, the method provides the first persistent objects sorted according to this condition. If you haven’t defined a sort condition, you shouldn’t make any assumptions of how the Query Service selects the returned subset of objects. If you don’t use the `I_UPTO` parameter or pass the 0 value, the number of returned objects won’t be limited. You shouldn’t pass negative values to the parameter.

- The `I_SUBCLASSES` parameter is only relevant if subclasses exist for the persistent class and if you’ve defined a type identifier in the persistence representation. If you pass the `abap_true` value to this parameter, the method also instantiates the objects of the persistent class, which simultaneously belong to a subclass of the persistent class. If a type identifier does exist, the method has a polymorphic behavior; that is, it will return objects of the subclasses automatically. Attributes that are defined in subclasses are also filled in this case, even if you’ve called the method in the class agent of the superclass. If the default setting, `abap_false`, is used and a type identifier is specified, the method only returns persistent objects that belong to the respective persistent class but not to a subclass.

  If no type identifier is defined, the method can’t decide efficiently if an object belongs to a subclass. In this case, it also provides a reference to an object of the persistent class that is managed by the class agent even if the object also belongs to a subclass.

One of the major differences between the three different procedures that enable you to instantiate multiple persistent objects via Object Services lies in the number of required database accesses:
The single instantiation process (refer to Listing 4.1) accesses the database to determine the keys of the persistent objects. Afterwards, the instantiation of each persistent object leads to another database access.

For mass instantiation (refer to Listing 4.2), ideally only two database accesses are necessary to instantiate multiple persistent objects: When you've determined the keys of the persistent objects with the first database access, the Persistence Service can often load all requested persistent objects from the database with only one additional database access. If there are numerous keys, it's possible that the ABAP runtime environment performs multiple database access and only loads a subset of the requested persistent objects from the database with each access.

You can reduce the number of database accesses to a minimum using the Query Service (refer to Listing 4.3). With only one database access, the Query Service passes the filter and sort conditions defined by you to the database system. As a result of this query, the database system returns the persistent attributes for all objects that correspond to this query so that no additional database access are necessary.

Due to the reduced communication requirement between application server and database system, you can further accelerate the instantiation process for persistent objects in many cases if you use the Query Service instead of mass instantiation. For example, the instantiation of 100,000 persistent objects using the Query Service can be twice as fast as with mass instantiation. Here as well, the factor by which the speed differs depends on the system and on the persistent class.

4.4 More Complex Selections Using the Query Service

The simple example from Section 4.3, Simple Selections Using the Query Service, illustrated how you can create a query with a plain filter condition using the Query Service. This section now discusses how you can formulate a more complex filter condition consisting of several individual criteria and define the sort condition.

Filter conditions also allow you to use *query parameters*. Query parameters enable you to create a query object with a filter condition once and then execute this query with different values in the individual criteria of the filter condition. For example, you can create a query object in whose filter condition you can define that you want to instantiate flight plans with a specific departure location and a
specific destination location. In the filter condition, you don’t specify the concrete airports, but refer to the query parameters first. You can then execute the query created in this way several times and specify through the query parameters which concrete departure location and destination location is supposed to be used for this execution of the query.

### 4.4.1 Defining Query Parameters

Without you having to make further specifications, each query provides three query parameters, which you can use to formulate filter conditions. These are called \( \text{PAR1}, \text{PAR2}, \text{and PAR3} \). If you want to use more than three query parameters in the filter condition or assign meaningful names to the query parameters, you have the following two options:

- You pass the \_PARAMETERS importing parameter to the CREATE_QUERY method when creating the query. To do so, you use a string in which the individual names of the query parameters are separated by a blank. Listing 4.4 shows the definition of two query parameters called \( \text{PAR\_AIRPFROM} \) and \( \text{PAR\_AIRPTO} \). For this way of defining query parameters, case sensitivity isn’t relevant. Irrespective of whether you use only lowercase letters or only uppercase letters to write the names of the parameters, the system creates query parameters with names in uppercase letters.

```plaintext
DATA: ri_query_manager TYPE REF TO if_os_query_manager,
      ri_query TYPE REF TO if_os_query.

ri_query_manager = cl_os_system=>get_query_manager( ).

* Create a new query and define the names of the parameters
* simultaneously
ri_query =
  ri_query_manager->create_query(
    i_parameters = 'PAR\_AIRPFROM PAR\_AIRPTO' ).

Listing 4.4 Setting the Parameter Names When Creating a Query
```

- Alternatively, you can define the names of the query parameters using a so-called parameter expression after you have created the query. For this purpose, you can have the query object provide a reference to an expression factory using the GET_EXPR_FACTORY method. The expression factory provides a method called CREATE_PARAMETERS_EXPR, which creates a parameter expression and
returns a reference to the object. You can then successively transfer the names of the query parameters to the `APPEND` method of the parameter expression with each call of the method. Finally, you pass the complete parameter expression with the names of all query parameters to the `SET_PARAMETERS_EXPR` method of the query.

When creating parameter expressions, case sensitivity is relevant for the names of the query parameters. You should only use uppercase letters for all parameter names to avoid problems during the execution of the query.

Like Listing 4.4, Listing 4.5 illustrates the described process by means of two parameters called `PAR_AIRPFROM` and `PAR_AIRPTO`.

```
DATA: ri_query_manager TYPE REF TO if_os_query_manager,
      ri_query TYPE REF TO if_os_query,
      ri_expr_factory TYPE REF TO if_os_query_expr_factory,
      ri_parameters_expr TYPE REF TO if_os_query_parameters_expr.

* Create a new query
  ri_query_manager = cl_os_system=>get_query_manager( ).
  ri_query = ri_query_manager->create_query( ).

* Get a reference to the expression factory
  ri_expr_factory = ri_query->get_expr_factory( ).

* Create a new parameter expression
  ri_parameters_expr = ri_expr_factory->create_parameters_expr( ).

* Define the names of the parameters in the parameter expression
  ri_parameters_expr->append( 'PAR_AIRPFROM' ).
  ri_parameters_expr->append( 'PAR_AIRPTO' ).

* Assign parameter expression to the query
  ri_query->set_parameters_expr( ri_parameters_expr ).
```

Listing 4.5 Setting the Parameter Names Using a Parameter Expression

Defining the parameter names via a parameter expression involves much more effort than defining the names during the creation of the query. Because the expressive power of the two variants is the same, you should always define parameter names during the creation of the query.
The names of the query parameters can consist of the letters A to Z, the numbers 0 to 9, and the underscore. The name must start with a letter. To avoid naming conflicts, the name of a parameter mustn’t be identical to the name of an attribute of the persistent class for which you want to instantiate the persistent objects.

You can use the defined parameters for the formulation of the filter condition. When executing the query, you then have to specify a value for any parameter that you used in the filter condition.

### 4.4.2 Defining the Filter Condition

The filter condition enables you to limit the objects a query is supposed to return via the Query Service. For this purpose, you can — referring to the persistent attributes of the persistent objects — specify criteria that are supposed to be met by any object that the query returns.

You can also execute a query using the Query Service without defining a filter condition. In this case, the query returns references to all persistent objects of the persistent class.

Similarly to parameter names, you have two options for defining filter conditions:

- You specify the filter condition in an SQL-similar syntax when creating the query.
- You define a so-called filter expression after creating the query.

#### Defining the Filter Condition During the Creation of the Query

When creating a query using the `CREATE_QUERY` method, you can work with the `I_FILTER` parameter to pass a string that contains the filter condition. To formulate filter conditions, you can use a subset of the basic language elements, which also enable you to define a `WHERE` clause in Open SQL. Also, an additional operator is available, which allows for using references to further persistent objects in conditions.

The relational operators, `=` (equal), `<>` (not equal), `<` (less than), `<=` (less than or equal), `>` (greater than), and `>=` (greater than or equal), enable you to define which values a persistent attribute may adopt so that the result of the query includes the persistent object. You always have to define the name of a persistent attribute
on the left side of the relational operator, but the right side can be the name of a persistent attribute, the name of a query parameter, or a value in the form of a literal. Literals with all data types, including numbers, always need to be enclosed in single inverted commas (') in the string that contains the filter condition.

Listing 4.6 shows three possible filter conditions with relational operators:

- **v_filter_1** selects all flight plans in which a maximum flight time of 120 minutes is specified. Here, as a literal, the maximum flight time is enclosed in inverted commas. When specifying the filter condition as a string literal in the ABAP source code, you have to use two single inverted commas, respectively, so that the string contains an inverted comma at the appropriate position.
- **v_filter_2** enables you to instantiate all flights that take place on a specific date. For this purpose, you must pass a concrete date value for the default `PAR1` query parameter when executing the query.
- **v_filter_3** compares two persistent attributes. You can use this filter condition to select all flights for which the same number of seats is occupied in the business class and in the first class.

```
DATA: v_filter_1 TYPE string,
    v_filter_2 TYPE string,
    v_filter_3 TYPE string.

v_filter_1 = 'FLTIME <= ''120'''.
v_filter_2 = 'FLDATE = PAR1'.
v_filter_3 = 'SEATSOCC_B = SEATSOCC_F'.
```

Listing 4.6 Filter Conditions with Relational Operators

The **LIKE** operator enables you to check the value of a persistent attribute against a template. In the template, you can use the underscore (_) if exactly one random character is possible at the corresponding position. If any number of characters can be used here, use the percentage sign (%). You can use the **ESCAPE** addition to define an escape character. If the selected escape character is used in front of the underscore or the percentage sign, these characters lose their special function. This way, you can express that the actual underscore or percentage sign is required at a specific position also when using the **LIKE** operator.

Listing 4.7 provides examples of the use of the **LIKE** operator:
More Complex Selections Using the Query Service

- v_filter_4 searches for all flight customers with email addresses that contain the characters "@sap.". The number of additional characters that precede these characters is unlimited, but there can be only two arbitrary characters after these characters.

- v_filter_5 searches for all customers with an email address that contains the name Benjamin followed by an underscore. Because the actual underscore sign is needed here and not the meaning of the underscore as a wildcard for any character, the number sign (#) is defined as the escape character. Written one after the other, the number sign escape character and the underscore effect that the query searches the email address for the underscore and doesn’t interpret the underscore as a wildcard for any character.

DATA: v_filter_4 TYPE string,
      v_filter_5 TYPE string.

v_filter_4 = 'EMAIL LIKE ''%@sap.__'''.
v_filter_5 = 'EMAIL LIKE ''%Benjamin#_%'' ESCAPE ''#'''.

Listing 4.7 Filter Conditions with the LIKE Operator

Null values, that is, fields of a database table that have explicitly been defined as empty, usually don’t occur if you work with Object Services in a database table. Only if you’ve created new fields for a database table that already contains data records or if you write to the database table directly with Open SQL and without Object Services, can it happen that a database table contains a null value — instead of the respective initial value of the data type. To identify this kind of constellation, you can use the IS NULL operator.

In addition to the operators described before, which are also available in Open SQL, you can also work with the EQUALSREF operator in the filter condition of a query when using the Query Service. The EQUALSREF operator enables you to check if a persistent reference refers to the same persistent object as a usual reference to a persistent object in the running application. Before the EQUALSREF operator, you always have to specify the name of a persistent attribute that is defined as a persistent reference in the persistence representation. The name of a parameter must always follow the operator.

You can use the logical operators AND and OR to merge two expressions into a new expression. The logical operator NOT enables you to invert the result of the
expression. You can use parentheses to define the sequence in which the system is supposed to evaluate the individual logical operators. In this case, every parenthesis needs to be separated by at least one blank from all other parts of the filter condition.

The query, which you can execute with the filter condition from Listing 4.8, instantiates all flight plans in which a departure airport is in Germany and a destination airport is in the United States or in Japan.

**DATA:** v_filter_6 TYPE string.

v_filter_6 =
    'COUNTRYFROM = ''DE'' AND ' &
    '( COUNTRYTO = ''US'' OR COUNTRYTO = ''JP'' )'.

**Listing 4.8 Filter Condition with Logical Operators**

If you define the filter condition when creating the query, case sensitivity isn’t relevant. When executing the query, the system automatically converts the names of parameters and attributes as well as the logical operators to uppercase. Case sensitivity is only important for literals against which you want to check the values of attributes.

**Defining the Filter Condition Using a Filter Expression**

Similarly to the definition of the names of the used query parameters, you can also use the expression factory to define the filter condition. Here, the expression factory enables you to generate a filter expression, which you finally assign to the query.

The basic idea of this kind of filter condition definition is that you create a tree of objects in which each object represents a part of the filter condition. Every leaf of a tree stands for an individual condition referring to a persistent attribute. The internal nodes link partial conditions by means of logical operators.

Four types of expression objects can be used as the leaves of the tree:

- **Operator expressions** enable you to compare a persistent attribute with another persistent attribute, with a parameter, or with a passed value using a relational operator. Here, you can use the same relational operators as for the definition of the filter condition during the creation of the query.
LIKE expressions allow you to check a persistent attribute against a template using the LIKE operator.

Checks for the null value are carried out using the IS NULL expression. These checks correspond to the use of the IS NULL operator.

Like the EQUALSREF operator, the reference expression determines if a persistent reference refers to a persistent object defined by you.

You create all expressions using the corresponding method of the expression factory. Every method provides individual parameters that you can use to transfer specifications such as the persistent attribute that is supposed to be compared, the relational operator, or the comparison value. In most of the methods, an importing parameter called I_IDX is defined. If you want to refer to a parameter of the query in an expression, use this importing parameter to specify the index of the query parameter. The index of the query parameter can be derived from the sequence in which you have defined the names of the query parameters. The first query parameter is assigned index 1; the Query Service numbers all other parameters in ascending order.

For several methods, you can either pass a value that the Query Service automatically encloses in inverted commas in the database query, or you can pass a value that already contains the required inverted commas. The names of the importing parameters for which you have to pass values including the inverted commas end with the _W_QUOTES (with quotes) suffix. If in doubt, you should always use the alternative importing parameter without the _W_QUOTES suffix.

AND and OR expressions enable you to link two expressions with a logical “and” or with a logical “or”. Because the linked expressions can be both individual expressions and a tree of already-created links of several expressions, you can use these expressions to create trees of any size. When creating a NOT expression, you only transfer a single expression. The NOT expression then negates this expression.

Figure 4.1 and Listing 4.9 show the same filter condition, which was already formulated in Listing 4.8. The UML object diagram in Figure 4.1 illustrates the objects the filter condition consists of. There is one object for each subexpression the filter condition contains. The tree as a whole describes the filter condition that states that the departure airport is supposed to be in Germany and the destination airport in the United States or in Japan.
Selecting Persistent Objects

Figure 4.1 Object Diagram of a Filter Expression

DATA: rf_ca_spfli TYPE REF TO /iot/ca_spfli.
    ri_query_manager TYPE REF TO if_os_query_manager.
    ri_query TYPE REF TO if_os_query.
    ri_expr_factory TYPE REF TO if_os_query_expr_factory.
    ri_filter_from_de TYPE REF TO if_os_query_filter_expr.
    ri_filter_to_us TYPE REF TO if_os_query_filter_expr.
    ri_filter_to_jp TYPE REF TO if_os_query_filter_expr.
    ri_filter TYPE REF TO if_os_query_filter_expr.
    ta_ro_spfli TYPE osreftab.

ri_query_manager = cl_os_system=>get_query_manager( ).
rf_ca_spfli = /iot/ca_spfli=>agent.

* Create query
ri_query = ri_query_manager->create_query( ).

* Get reference to expression factory
ri_expr_factory = ri_query->get_expr_factory( ).

* Create filter expression: departure country Germany
ri_filter_from_de =
    ri_expr_factory->create_operator_expr(
        i_attr1 = 'COUNTRYFR'
        i_operator = '='
        i_val = 'DE' ).
* Create filter expression: destination country USA
ri_filter_to_us =
   ri_expr_factory->create_operator_expr(
      i_attr1 = 'COUNTRYTO',
      i_operator = '='
      i_val = 'US' ).

* Create filter expression: destination country Japan
ri_filter_to_jp =
   ri_expr_factory->create_operator_expr(
      i_attr1 = 'COUNTRYTO',
      i_operator = '='
      i_val = 'JP' ).

* Create filter expression: "or" link between destination countries
   ri_filter_to =
   ri_expr_factory->create_or_expr( i_expr1 = ri_filter_to_us, i_expr2 = ri_filter_to_jp ).

* Create filter expression: "and" link between departure country and the two alternative destination countries
   ri_filter =
   ri_expr_factory->create_and_expr( i_expr1 = ri_filter_from_de, i_expr2 = ri_filter_to ).

* Assign complete filter expression to query
   ri_query->set_filter_expr( ri_filter ).

* Execute query
   ta_ro_spfli =
   rf_ca_spfli->if_os_ca_persistency-get_persistent_by_query( ri_query ).

Listing 4.9 Definition and Use of a Filter Expression

To create the individual objects that are supposed to form the tree with the filter condition, you first require a reference to the expression factory again. In the expression factory, you call the respective methods, one after the other, for creating the specific expression objects.
For more complex filter conditions that consist of multiple objects (see Listing 4.9), you have to create the tree according to the bottom-up approach: You first generate the tree’s leaves, that is, the conditions that directly refer to a persistent attribute. The names of the attributes must be transferred in uppercase characters only. Then, you create the internal nodes, which link the already-existing objects by using the logical operators **AND**, **OR**, and **NOT**. Finally, you assign the query as a filter condition to the root of the tree (in Figure 4.1, the highest-level object) via its `SET_FILTER_EXPR` method. Because only the root is connected to all parts of the tree via the references, it’s the only object from which the Query Service can derive the complete filter condition.

**Comparing the Two Variants for the Definition of the Filter Condition**

The expressive power of the two variants for the definition of filter conditions is the same; that is, you could also use the respective other variant to formulate any filter condition that you can create using one of the two variants. However, you should create filter conditions during the creation of the query, because — compared to using a filter expression — this can be implemented with less effort, and the source code is more legible. It only makes sense to use the filter expression if you want to compose the filter condition dynamically at runtime.

**4.4.3 Defining the Sort Condition**

The sort condition of a query enables you to specify in which sequence the persistent objects are supposed to be sorted in the result table after the query has been executed. For this purpose, you can refer to one or several persistent attributes and define for each attribute whether the sorting is supposed to be done in ascending or descending order.

If you don’t define a sort condition, Object Services return the persistent objects in the sequence in which the database system returned the corresponding data records. In many cases, the data records are sorted in ascending order according to the primary key of the database table. However, because you can’t be sure that every system has this behavior, you should always explicitly specify the sort condition if how the persistent objects are sorted is relevant for your application.
You can define the sort condition in two ways:

- You pass the \texttt{I\_ORDERING} parameter to the \texttt{CREATE\_QUERY} method during the creation of the query.
- You define a sort expression via the expression factory and pass it to the \texttt{SET\_ORDERING\_EXPR} method of the query.

### Defining the Sort Condition During the Creation of the Query

To the \texttt{I\_ORDERING} parameter of the \texttt{CREATE\_QUERY} method, you have to transfer a string that contains the name of a persistent attribute followed by the \texttt{ASCENDING} (sorted in ascending order) or \texttt{DESCENDING} (sorted in descending order) keyword. The persistent attributes and the keywords for the sorting sequence must be separated by blanks, respectively. For this variant, case sensitivity isn’t relevant.

The sequence in which you specify multiple persistent attributes determines the sequence in which the system sorts the individual attributes: The system initially sorts the persistent object by the first specified persistent attribute. If required and defined, it then sorts all other persistent attributes.

Listing 4.10 shows an example of a sort condition with two persistent attributes. The query that is created in the listing instantiates flights from the flight data model. Object Services sort the flights in descending order according to the flight date. Consequently, earlier flights are listed higher in the result table than later flights. The query sorts all flights on the same date in descending order according to the \texttt{SEATSMAX} attribute. So, it first returns the flights with the most seats in the economy class and lastly the flights with the lowest capacity.

```cpp
gi_query =
gi_query_manager->create_query(
    i_ordering = 'FLDATE ASCENDING SEATSMAX DESCENDING').
```

**Listing 4.10** Defining the Sort Condition During the Creation of the Query

### Defining the Sort Condition Using a Sort Expression

Similar to parameter expressions and filter expressions, you can also create a query, in which no sort condition is defined, and then use the expression factory to create a sort expression.

You can call two methods on a sort expression: a method for adding a persistent attribute according to which the sorting is to be done in ascending order (\texttt{APPEND_}
ASCENDING), and a method for adding a persistent attribute according to which the sorting is to be done in descending order (APPEND_DESCENDING). You always have to transfer the name of the persistent attribute in uppercase letters. Here, the sequence in which you add the persistent attributes also determines the sequence in which the persistent attributes will be used to determine the order of the persistent objects.

Listing 4.11 defines the same sort condition as Listing 4.10 using a sort expression. For this purpose, the CREATE_ORDERING_EXPR method of the expression factory is called. It creates a sort expression. The respective methods for adding persistent attributes by which the persistent objects will be sorted in descending and ascending order then are called on the sort expression. Object Services are supposed to sort the query result based on these attributes. Finally, the sort expression is assigned to the query.

```
DATA: ri_query_manager TYPE REF TO if_os_query_manager,
      ri_query TYPE REF TO if_os_query,
      ri_expr_factory TYPE REF TO if_os_query_expr_factory,
      ri_ordering_expr TYPE REF TO if_os_query_ordering_expr.

ri_query_manager = cl_os_system=>get_query_manager( ).

* Create query
   ri_query = ri_query_manager->create_query( ).

* Get reference to expression factory
   ri_expr_factory = ri_query->get_expr_factory( ).

* Create sort expression
   ri_ordering_expr = ri_expr_factory->create_ordering_expr( ).

* Define persistent attributes by which the objects are sorted:
  * - Ascending by flight date
  * - Descending by maximum seating in economy class
   ri_ordering_expr->append_ascending( 'FLDATE' ).
   ri_ordering_expr->append_descending( 'SEATSMAX' ).

* Assign sort expression to query
   ri_query->set_ordering_expr( ri_ordering_expr ).
```

**Listing 4.11** Defining the Sort Condition Using a Sort Expression
Comparing the Two Variants for the Definition of the Sort Condition

The expressive power of the two variants for the definition of the sort condition is identical, too. In general, you can always use both variants to formulate any kind of sort condition. However, the variant for defining the sort condition during the creation of the query is usually more comfortable and less complex, particularly for constant sort conditions. The variant for the definition via a sort expression might be more useful if you want to create the sort condition dynamically at runtime.

4.4.4 Passing Concrete Values for Query Parameters

For all query parameters that you have defined and actually used in the filter condition, you have to transfer the concrete value with which you want to execute the query when calling the query. Only if you haven't made any reference to a query parameter in the filter condition, can you execute a query without having to specify values for query parameters.

There are two options for passing parameter values, namely, a simplified option with limited functions, and an option for more complex queries:

- If you use three parameters at the most and don’t evaluate persistent references in the query, you can pass the values of the query parameters to the `IF_OS_CA_PERSISTENCY~GET_PERSISTENT_BY_QUERY` method using the `I_PAR1` to `I_PAR3` importing parameters.

- You can transfer an internal table with data references to the `IF_OS_CA_PERSISTENCY~GET_PERSISTENT_BY_QUERY` method in any case, that is, irrespective of the number of the parameters and also if you work with persistent references.

Passing Individual Parameters: `I_PAR1` to `I_PAR3`

The parameter interface of the `IF_OS_CA_PERSISTENCY~GET_PERSISTENT_BY_QUERY` method contains three optional importing parameters called `I_PAR1`, `I_PAR2`, and `I_PAR3`. You can use these parameters to transfer concrete values for the parameters used. Because the importing parameters are defined with the `ANY` type, you can pass any value of an elementary type to them. Passing a reference to a persistent object, in contrast, isn’t possible.

Listing 4.12 instantiates all flights that take place in October 2009. Here, the filter condition refers to the default parameters, `PAR1` and `PAR2`. The concrete values
(October 01 and October 31, 2009, in this case) aren't passed until the query is being executed. Without having to create a new query object, you can execute the query any number of times and always request flights from a different date interval.

```
DATA: ri_query_manager TYPE REF TO if_os_query_manager,
     ri_query   TYPE REF TO if_os_query,
     ta_ro_flights TYPE osreftab,
     rf_ca_sflight TYPE REF TO /iot/ca_sflight.

rf_ca_sflight = /iot/ca_sflight=>agent.
ri_query_manager = cl_os_system=>get_query_manager( ).

* Create query
ri_query =
    ri_query_manager->create_query(
        i_filter = 'FLDATE >= PAR1 AND FLDATE <= PAR2' ).

* Execute query
   ta_ro_flights =
    rf_ca_sflight->if_os_ca_persistency~get_persistent_by_query(
        i_query = ri_query
        i_par1  = '20091001'
        i_par2  = '20091031' ).
```

**Listing 4.12  Query Passing Individual Parameters**

To use the I_PAR1 to I_PAR3 importing parameters when executing a query, you don’t necessarily have to work with query parameters called PAR1 to PAR3. The I_PAR1 to I_PAR3 importing parameters, respectively, refer to the first, second, and third defined query parameter, irrespective of its name. For example, if you have defined two query parameters with user-defined names, you can assign concrete values to them using the I_PAR1 and I_PAR2 importing parameters when executing the query.

**Passing an Internal Table Using Query Parameters: I_PARAMETER_TAB**

The I_PARAMETER_TAB importing parameter provides an alternative for passing values for query parameters. This importing parameter enables you to transfer an internal table in which each line contains a data reference to the value to which you want to pass the query parameter. A data reference allows for referring to a variable, a constant, a literal of an elementary type, or a reference to an object.
This variant for passing query parameters also doesn’t require a direct reference to the names of the query parameter. The sequence in which you defined the names of the query parameters also determines the sequence you have to use for the data references in the internal table: The data reference to the value for the first defined query parameter needs to be specified in the first line, the data reference to the value for the next defined parameter in the second line, and so on.

Listing 4.13 illustrates how you can populate an internal table with data references and then use this table to execute a query.

```
DATA: dr_rf_airport TYPE REF TO data,
     rf_ca_airport TYPE REF TO /iot/ca_sairport,
     rf_ca_counter TYPE REF TO /iot/ca_scounter,
     rf_airport TYPE REF TO /iot/cl_sairport,
     ri_query_manager TYPE REF TO if_os_query_manager,
     ri_query TYPE REF TO if_os_query,
     ta_parameters TYPE osdreftab,
     ta_ro_counters TYPE osreftab.

rf_ca_airport = /iot/ca_sairport=>agent.
rf_ca_counter = /iot/ca_scounter=>agent.
ri_query_manager = cl_os_system=>get_query_manager( ).

* Create query
  ri_query =
    ri_query_manager->create_query( 
      i_filter = 'RF_AIRPORT EQUALSREF PAR1').

* Instantiate Frankfurt Airport
  rf_airport = rf_ca_airport->get_persistent( i_id = 'FRA' ).

* Append airport object data reference to parameter table
  GET REFERENCE OF rf_airport INTO dr_rf_airport.
  APPEND dr_rf_airport TO ta_parameters.

* Execute query
  ta_ro_counters =
    rf_ca_counter->if_os_ca_persistency~get_persistent_by_query( 
      i_query = ri_query
      i_parameter_tab = ta_parameters ).
```

**Listing 4.13** Query with Transfer of an Internal Table with Parameters
The filter condition of the query contains the `EQUALSREF` operator, that is, a condition referring to a persistent reference. In this case, the query is supposed to determine all airline sales counters at a specific airport. For this purpose, the filter condition refers to the `RF_AIRPORT` attribute. For each counter, this attribute contains a persistent reference to the airport at which the counter is located.

To use a condition with a persistent reference in a query, you need a reference to an already-instantiated persistent object. Consequently, the corresponding persistent object is instantiated to determine the counters at Frankfurt Airport.

The `GET REFERENCE OF` statement generates a data reference, which you require for the internal table with the values of the query parameters. If you apply this to the reference to a persistent object, you receive a two-level reference, that is, a data reference to a reference to a persistent object in this case. Then, you have to append this data reference to the internal table that contains the values of the query parameters before passing the table to the `I_PARAMETER_TAB` importing parameter during the execution of the query.

**Comparing the Two Variants for Passing Concrete Values for Query Parameters**

It’s considerably easier to pass individual parameters, `I_PAR1` to `I_PAR3`, than the internal table with the `I_PARAMETER_TAB` parameters. You should therefore only use the internal table if a transfer of individual parameters is out of question, that is, if more than three parameters are used or a persistent reference is referenced.

### 4.5 Comparing the Query Service and Open SQL

You can benefit from two advantages if you work with a simple query using the Query Service to instantiate persistent objects instead of determining the keys of persistent objects with a `SELECT` query in Open SQL and then loading them: The ABAP source code that you need to implement is more compact, and the whole process requires only one database access, which reduces the load on the application server and database system as well as the communication between them. This, in turn, may make your application significantly faster.

If you use the Query Service, the keys of the objects on the application server aren’t known before the objects are instantiated. Consequently, you can’t work with the keys of the objects before the instantiation. As further described in Chapter 8,
Integration of the SAP Lock Concept and Object Services, this is a disadvantage, particularly in the SAP Lock Concept context.

The expressive power of the queries using the Query Service and the expressive power of the SELECT queries in Open SQL differ considerably. The Query Service always instantiates complete persistent objects. Therefore, SELECT clauses for the specification of the fields that are supposed to be loaded, equivalents to the GROUP BY and HAVING clauses, and aggregate functions don’t exist. The functional scope of the Query Service also doesn’t include subqueries or joins for arbitrary tables.

A FROM clause isn’t required if you work with the Query Service because the Query Service uses the information from the persistence representation, which contains the underlying database tables. Concerning persistent references, the Query Service also uses already-existing information in the system and thus reduces the development work compared to an equivalent query in Open SQL.

Open SQL is an integral part of the ABAP programming language. As a result, the system can check the syntax of queries in Open SQL already during the development time if they don’t contain dynamic components. The syntactic correctness of a filter condition of a query that was implemented using the Query Service, in contrast, can be checked by the system at runtime only.

Both the selection via the Query Service and the selection with Open SQL are solely based on the database’s data. If you’ve created, changed, or deleted persistent objects in the running program, and you don’t want to transfer these changes to the database yet, the result of the selection is still based on the original state without considering the changes. The next section describes how you can use the functions of the class agents to have the selections consider changes that you’ve made to persistent objects only in the memory of the running application.

## 4.6 Handling Newly Created and Changed Objects

Each class agent of a persistent class contains methods that you can respectively use to determine the persistent objects of the class that have a certain management state. This way, you can among other things determine all persistent objects of the class that you’ve created, changed, or deleted in the current top-level transaction.

The methods are defined in the IF_OS_CA_INSTANCE interface. Their respective names begin with the GET_prefix, which is then, except for one exception, followed by the
name of the corresponding management state. This exception is the \texttt{GET\_CREATED} method, which determines the persistent objects in the \texttt{NEW} management state.

The respective methods only provide objects that belong to the respective persistent class that the class agent manages. The methods don't return objects that are included in the management of a class agent of a subclass or superclass.

Listing 4.14 shows how you can adapt the result of a query in such a way that it also considers newly created objects. The query determines all flights that take place on the current date. When the query has been executed, the \texttt{GET\_CREATED} method determines all references to newly created objects. In a \texttt{LOOP} loop, it's then manually checked for every newly created flight if it meets the filter condition. If so, the \texttt{GET\_FLDATE} access method reads the flight date and compares it to the current date. If the flight date is identical to the current date, the newly created flight is added to the internal table that contains all previously determined flights on the current date.

\begin{verbatim}
DATA: rf_ca_sflight  TYPE REF TO /iot/ca_sflight,
       rf_sflight   TYPE REF TO /iot/cl_sflight,
       ri_query_manager  TYPE REF TO if_os_query_manager,
       ri_query     TYPE REF TO if_os_query,
       ro_sflight   TYPE REF TO object,
       ta_ro_sflights TYPE osreftab,
       ta_ro_created TYPE osreftab.

rf_ca_sflight = /iot/ca_sflight=>agent.
ri_query_manager = cl_os_system=>get_query_manager( ).

* Create query
ri_query =
  ri_query_manager->create_query(
    i_filter = 'FLDATE = PAR1' ).

* Execute query
ta_ro_sflights =
  rf_ca_sflight->if_os_ca_persistency~get_persistent_by_query(
    i_query = ri_query
    i_par1  = sy-datum ).

* Determine newly created persistent objects
ta_ro_created =
  rf_ca_sflight->if_os_ca_instance~get_created( ).
\end{verbatim}
* Check every newly created persistent object against
* filter condition

```abap
LOOP AT ta_ro_created INTO ro_sflight.
  * Typecast: OBJECT -> /IOT/CL_SFLIGHT
  rf_sflight := ro_sflight.

  IF rf_sflight->get_fldate( ) = sy-datum.
  * Object corresponds to filter condition
  * Add to result table
  APPEND rf_sflight TO ta_ro_sflights.
  ENDIF.
ENDLOOP.

Listing 4.14 Manual Consideration of Newly Created Objects
```

To also include changed objects in a selection, you have to consider two constellations:

- **Owing to the change, a persistent object can correspond to the filter condition while the present state in the database doesn't meet the filter condition yet.** Consider this case in the same way as in Listing 4.14 and determine the changed objects using the `GET_CHANGED` method. In addition, remember that a changed object might have already met the filter condition previously. You therefore need to ensure that the result table doesn't add an already-contained object again.

- **In the reverse case, the present state in the database meets the filter condition, but the changed objects in the memory of the running application no longer meets the condition.** To remove such objects from the query's result, you should initially check the management state of any determined object (`IF_OS_CA_INSTANCE~GET_STATUS` method, see Section 3.4, Management States of Persistent Objects, in Chapter 3). You have to check all objects in the `CHANGED` management state manually against the filter condition using the access methods. If an object no longer meets the filter condition, remove it from the result.

Object Services automatically consider objects that were deleted in the running program to a certain degree. If you try to use mass instantiation or a query via the Query Service to instantiate an object that has already been deleted in the running program, the system raises an exception of the `CX_OS_OBJECT_NOT_FOUND` class. As a result, the deleted object automatically isn't instantiated. If the exception occurs, the instantiation of further objects is also terminated.
That means you don’t receive a result that doesn’t include the deleted objects, you receive no result at all. To still instantiate all objects that haven’t been deleted — despite existing deleted objects — you can use an approach as implemented in Listing 4.1: Determine the keys of the objects by means of a query in Open SQL and then instantiate them separately. This way, you only include the persistent objects in the result during whose instantiation no exception of the `CX_OS_OBJECT_NOT_FOUND` class occurs.

As of EhP2 for Release 7.0 of SAP NetWeaver AS ABAP, you can also control the Query Service’s behavior regarding deleted objects. If you pass the `IGNORE_DELETED` option when executing a query, the Query Service doesn’t raise an exception when it detects deleted objects. Instead, the query provides a result that doesn’t include the deleted objects.

In more complex applications that change a lot of persistent objects during a top-level transaction, you usually have to consider newly created and changed objects in the selection of persistent objects. If your application allows for the execution of queries when you’ve already changed persistent objects in the current top-level transaction, the queries don’t provide a result that is correct with regard to content if newly created and changed objects aren’t considered. However, if your application is structured in such a way that every top-level transaction begins with loading all involved objects first and then makes all changes, special handling for newly created and changed objects isn’t required in the selection.

4.7 Summary

This chapter described how you can select persistent objects according to user-defined criteria. In some cases, you can use the Object Services Query Service for this purpose. You also learned in which cases it’s necessary and useful to use Open SQL statements that have already been available in classic ABAP instead of the Query Service.

This chapter also provided a first proposal for an enhancement of the Object Services functionality: considering objects in the memory for selections. The next chapter explains the internal functioning of Object Services and thus introduces the basic principles of the additional enhancements, which are discussed in Chapters 6 to 8.
Index

_CLASS;scope, 185

A
ABAP Debugger, 172
Abstract class, 45, 117, 184
Activation, 26, 30, 125, 172
ACTIVE_CLASS_AGENT, 135, 140
AGENT, 31, 32, 68, 119, 124
ALV grid control, 151
AND, 103
APPEND_ASCENDING, 108
APPEND_DESCENDING, 108
Append structure, 26
ASCENDING, 107
Assignment type, 24, 40, 47, 124
Asynchronous update, 71, 73, 74
Attribute
  Persistent, 29
  Transient, 29, 56, 84, 125, 146, 153, 154,
    161, 177
  Type, 25, 41, 43

B
Base agent, 117, 124, 149, 194
Business key, 24, 26, 29, 31, 33, 52, 53, 54,
  55, 89, 91, 120, 146, 149, 193

C
CALL FUNCTION IN UPDATE TASK, 72
Casting operator, 32, 33, 34, 52
Changeability, 24, 57, 146
CHANGED, 82, 87, 115, 138, 153, 198, 200
Check agent, 74, 77, 79, 126, 130, 192, 196,
  197, 199, 204
CL_ABAP_WEAK_REFERENCE, 128

Class
  Abstract, 45, 117, 184
    Persistent, 13, 19, 20, 21, 26, 27, 29, 30,
      37
Class agent, 30, 32, 34, 36, 38, 47, 51, 53, 54,
  55, 57, 65, 80, 81, 82, 89, 91, 94, 113, 117,
  121, 124, 127, 132, 134, 137, 140, 149, 165,
  170, 176, 179, 188, 190, 192, 193, 197, 202,
  204, 206, 210
Class Builder, 20, 21, 170, 172, 176
Class constructor, 68, 119, 124
Class GUID, 39, 40, 42, 47, 123, 135
Class Identifier, 24, 40
Class initializer, 69
CL_OS_CA_COMMON, 118
CL_OS_STATE, 132
  SET_OBJECT_FROM_STATE, 133
  SET_STATE_FROM_OBJECT, 133
CL_OS_SYSTEM, 61, 67, 73, 94, 117, 122,
  134, 140
  ACTIVE_CLASS_AGENT, 135
  EXTERNAL_COMMIT, 136
  GET_CLASS_AGENT, 134
  GET_CLASS_AGENT_INFO, 135
  GET_QUERY_MANAGER, 94
  GET_TRANSACTION_MANAGER, 61
  INIT_AND_SET_MODES, 67, 73
  INIT_STATE, 136
CL_OS_TRANSACTION, 129, 200
  CHAINED, 130
  CHECK_AGENTS, 130
  DATA_SAVE_STATE, 130
  FLAG_UPDT_MODE_CHANGED, 131
  SUBTRANSACTION, 130
  SUPER_TRANSACTION, 130
  TRANSACTION_STATE, 131
  UNDO_RELEVANT, 131
  UPDATE_MODE, 131
CL_OS_TRANSACTION_MANAGER, 129
  CURRENT_TRANSACTION, 129
  TOP_TRANSACTION, 129
CL_SYSTEM_UUID, 27
Index

COMMIT WORK, 59, 64, 72, 74, 78, 87, 131, 185
Compatibility mode, 64, 72, 74, 77, 80, 87, 131, 136, 185, 199
Concatenate method, 62, 66, 77, 80, 87
Constructor, static, 69
CREATED, 114
CREATE_PERSISTENT, 54, 55, 81, 84
CREATE_STRUCTURE, 148
CREATE_TRANSIENT, 56, 81
Current transaction, 63, 129
CX_DYNAMIC_CHECK, 35, 55, 64
CX_NO_CHECK, 36, 67, 74
CX_OS_CHECK_AGENT_FAILED, 76, 193, 204
CX_OS_DB, 74
CX_OS_NO_IMPLEMENTATION, 36, 55
CX_OS_OBJECT_EXISTING, 55
CX_OS_OBJECT_NOT_FOUND, 35, 37, 42, 57, 92, 115
CX_OS_SYSTEM, 67
CX_OS_TRANSACTION, 64, 74
CX_SY_DYN_CALL_ILLEGAL_METHOD, 143
CX_SY_MOVE_CAST_ERROR, 43
CX_SY_NO_HANDLER, 43

D

Data
Persistent, 13
Transient, 13
Database LUW, 59, 182
Database status, 120
DELETED, 82, 87, 92, 116, 138, 200
DELETE_PERSISTENT, 81, 83
DESCENDING, 107
Design pattern, singleton, 31, 69, 119
Direct update, 71, 73, 74, 185, 189
Downcast, 34, 92, 179, 198

E

Enhancement, 169, 176, 178
implementation, 171
Implementation, 169

F

Filter
Condition, 93, 96, 97, 99, 109, 112, 114, 179, 197, 210
Expression, 99, 102
Flight data model, 16, 31, 34, 39, 46, 89, 107, 120, 149, 152, 181
Function module
SYSTEM_UUID_CREATE, 27

G

Garbage collection, 117, 119, 127, 140
Generator setting, 85, 124, 125
GET_PERSISTENT, 34, 36, 83
Globally Unique Identifier (GUID), 26, 27

H

HANDLER_CHANGED, 197, 200
Horizontal mapping, 44, 45, 47

I

IF_OS_CA_INSTANCE, 80, 113, 122, 202
GET_STATUS, 80, 202
IF_OS_CA_PERSISTENCY, 31, 33, 36, 89, 91, 94, 109, 121, 149
GET_PERSISTENT_BY_KEY, 33, 36, 89, 121, 149
GET_PERSISTENT_BY_KEY_TAB, 91, 121
Index

GET_PERSISTENT_BY_OID, 31, 36
GET_PERSISTENT_BY_OID_TAB, 92
GET_PERSISTENT_BY_QUERY, 94, 109
IF_OS_CHECK, 75, 126, 198, 199
IS_CONSISTENT, 75, 199
IF_OS_FACTORY, 51, 52, 53, 55, 81, 82, 121, 138, 140, 149, 179, 196
CREATE_PERSISTENT, 52
CREATE_PERSISTENT_BY_KEY, 53, 55, 121, 149
DELETE_PERSISTENT, 81, 83
REFRESH_PERSISTENT, 82, 138, 179, 196
RELEASE, 82, 84, 140
IF_OS_FACTORY~CREATE_TRANSIENT, 56
IF_OS_FACTORY~CREATE_TRANSIENT_BY_KEY, 56
IF_OS_INSTANCE_MANAGER, 122
IF_OS_PERSISTENCY_MANAGER
GET_PERSISTENT_BY_QUERY, 123
IF_OS_STATE, 124, 132, 179
GET, 132
INIT, 126
INVALIDATE, 126, 179
SET, 132
IF_OS_TRANSACTION, 129
END, 61, 63, 65, 75, 77, 79, 86, 185
END_AND_CHAIN, 62, 80, 87, 130
GET_STATUS, 78
REGISTER_CHECK_AGENT, 75
START, 61, 79
UNDO, 61, 63, 65, 76, 77, 79, 86, 132, 185, 199
UNDO_AND_CHAIN, 62, 80, 87, 130, 132
IF_OS_TRANSACTION_MANAGER, 129
CREATE_TRANSACTION, 61, 79
GET_CURRENT_TRANSACTION, 63
GET_TOP_TRANSACTION, 64
IGNORE_DELETED, 116
Initialization block, static, 69
INIT_STATE, 136
Instance GUID, 26, 29, 31, 33, 35, 38, 39, 40, 43, 52, 53, 54, 55, 89, 91, 119, 124, 147, 195
Instantiation, 20, 30
Is-a relationship, 44
IS NULL, 103

L

Lazy loading, 37, 42, 161, 176
LIKE, 100, 103
LOADED, 82, 84, 86, 122, 128, 138, 141, 193
Local update, 71, 74
Lock
Conversion, 187, 189, 192, 195, 197, 199, 205, 206
Exclusive, 187, 189, 192, 195, 197, 199, 204, 205
Optimistic, 187, 189, 192, 193, 196, 197, 199, 205
Lock concept, 181
Locking strategy, 186
Optimistic, 186, 189, 192, 193, 195, 197, 200, 204, 206
Pessimistic, 186, 187, 191, 195, 204, 205
Lock mode, 186, 195
Logical Unit of Work (LUW), 59

M

Management state, 35, 51, 80, 113, 119, 127, 132
Mapping
Horizontal, 44, 45, 47
Object-relational, 11
Vertical, 44, 46, 47
Mapping Assistant, 21
Mapping, object-relational, 20, 161, 165, 167, 175, 195, 209
MOVE-CORRESPONDING, 141, 144
Multiple-table mapping, 28, 44

N

NEW, 79, 81, 87, 114, 138, 200
NOT, 103
NOT LOADED, 81, 82, 86, 119, 126, 127, 138, 140, 193, 205
NULL, 93, 101, 103, 128, 130
Number range object, 27, 55

215
# Index

## O

Object
- **Persistent**, 13, 19, 26, 30, 37, 51
- **Transient**, 35, 55, 82, 86, 122

**OBJECT_INFO**, 119, 128

**Object Navigator**, 20

**Object-oriented transaction**, 60, 86

**Object-oriented transaction mode**, 64, 69, 87, 131, 185, 198

**Object Reference**, 24, 40

**Object-relational mapping**, 11, 20, 161, 165, 167, 175, 199

**OBJECT_TO_STRUCTURE**, 142

**OO transaction**, 69

**OO transaction model**, 69

**Open SQL**, 89, 91, 99, 112, 116, 119, 153, 184, 195, 210

**Operator expression**, 102

**Optimistic lock**, 187, 189, 192, 193, 196, 197, 199, 205

**Optimistic locking**, 186, 189, 192, 193, 195, 197, 200, 204, 206

**OR**, 103

**OSCON**, 73, 78, 80, 202

**OS_GUID**, 25, 26, 39, 47, 92, 147

**OSREFTAB**, 92

**OSTYP_CA_INFO**, 135

## P

**Parameter expression**, 97

**Persistence representation**, 21, 22, 26, 29, 30, 31, 33, 34, 36, 38, 39, 40, 44, 47, 51, 53, 55, 57, 85, 92, 94, 101, 113, 118, 124, 146, 199, 162, 167, 193

**Persistence Service**, 13, 19, 23, 26, 28, 29, 30, 34, 35, 36, 37, 39, 41, 42, 43, 46, 47, 51, 52, 53, 54, 55, 56, 58, 60, 62, 63, 66, 72, 80, 83, 84, 86, 91, 96, 117, 118, 124, 127, 137, 147, 149, 162, 163, 165, 168, 177, 194, 196

**Persistent attribute**, 29

**Persistent class**, 13, 19, 20, 21, 26, 27, 29, 30, 37

**Persistent data**, 13

**Persistent object**, 13, 19, 26, 30, 37, 51

**Persistent reference**, 25, 27, 38

**Pessimistic locking**, 186, 187, 191, 195, 204, 205

**Plausibility check**, 153, 161, 176

**Polymorphism**, 49

**Prefix**
- **DEQUEUE_**, 184
- **ENQUEUE_**, 184
- **GET_**, 23, 37, 113, 143, 144, 163, 175
- **/IOT/**, 17
- **OSCON_DMODE_**, 73
- **OSCON_OSTATUS_**, 80
- **OSCON_SSTATUS_**, 131
- **OSCON_TSTATUS_**, 78
- **SET_**, 57, 145, 163, 175

**Private**, 24

**Protected**, 24

**Public**, 24, 94

## Q

**Query**, 90, 93, 94, 96

**Query Manager**, 94, 122

**Query parameter**, 96, 97, 100, 102, 109, 110

**Query Service**, 13, 91, 93, 96, 112, 115, 122, 152, 157, 179, 196, 210

## R

**Read Only**, 24, 146

**Reference**
- **Expression**, 103
- **Persistent**, 25, 27, 38
- **Strong**, 127
- **Usual**, 127
- **Weak**, 127

**REFRESH_ALL_OBJECTS**, 139

**REFRESH_OBJECTS_BY_AGENT**, 139

**REFRESH_PERSISTENT**, 82, 138, 179, 196

**Relationship**, is-a, 44

**RELEASE**, 82, 84, 140

**Representative object**, 37, 41, 42, 57, 62, 81, 84, 85, 87
Index

ROLLBACK WORK, 59, 61, 64, 87, 185
RTTC, 143, 147, 149
RTTI, 143
RTTS, 143
Run Time Type Creation, 143, 147, 149
Run Time Type Identification, 143
Run Time Type Services, 143

S

SAP Control Framework, 150
SAP Help Portal, 21
SAP Lock Concept, 14, 113, 182
SAP LUW, 58, 59, 61, 64, 72, 77, 80, 83, 86, 185, 187
SAP NetWeaver Application Server ABAP, 11, 17, 19, 59, 125, 141, 176, 181, 182, 210
       Release 6.10, 19, 35, 175
       Release 6.40, 147
       Release 6.40 SP16, 85
       Release 7.0, 90, 169, 175
       Release 7.0 EhP2, 27, 29, 33, 85, 116
       Release 7.0 SP6, 85
       Release 7.0 SP13, 92
SE24, 20
SE80, 20
SE93, 69
SEOCLSKEY, 123
SET_MODE_UNDO_RELEVANT, 77
SET_UPDATE_TASK_LOCAL, 71
Singleton, 30, 69, 119
Sort condition, 93, 96, 106, 107, 157
Sort expression, 107
SPECIAL_BKEY_TAB, 121
SPECIAL_OBJECT_INFO, 121
SPECIAL_OID_TAB, 121
STATE_WRITE_ACCESS, 85
Static constructor, 69
Static initialization block, 69
Strong reference, 127
STRUCTURE_TO_OBJECT, 144
Subordinate transaction, 130
Subtransaction, 63, 66, 77, 86, 129, 199
Suffix
     _W_QUOTES, 103
Superordinate transaction, 63, 130
Synchronous update, 71, 73, 74
System state, 136
SYSTEM_UUID_CREATE, 27

T

Top-level transaction, 61, 63, 64, 73, 77, 80, 83, 84, 86, 113, 126, 129, 159, 185, 188, 189, 193, 197, 199, 205
Transaction, 58
   Concatenate, 61
   Concatenation, 130
   Current, 63, 129
   Object-oriented, 60, 86
   SE24, 20
   SE80, 20
   Subordinate, 130
   Superordinate, 63, 130
Transaction code, 59, 69, 73
   SE93, 69
Transaction Manager, 60, 63, 67, 79, 122, 129, 159, 198
Transaction mode, 64, 69, 136, 158
   Compatibility mode, 64
   Object-oriented, 64, 69, 87, 131, 185, 198
Transaction model, 69
Transaction Service, 13, 51, 52, 58, 129
Transaction status, 78, 131
TRANSIENT, 82, 86, 141
Transient attribute, 29, 56, 84, 125, 146, 153, 154, 161, 177
Transient data, 13
Transient object, 35, 55, 82, 86, 122
TYP_BUSINESS_KEY, 121, 149
Type group, 73
Type identifier, 24, 47, 95, 147
Type object, 143, 145, 148
Type, OS_GUID, 25, 26

U

Undo mechanism, 77, 79, 86, 131
Universally Unique Identifier (UUID), 27
## Index

**Update**
- Asynchronous, 71, 73, 74
- Direct, 71, 73, 74, 185, 189
- Local, 71, 74
- Synchronous, 71, 73, 74
Update mode, 70, 73, 131
Usual reference, 127

**W**
- Weak reference, 127
- Web Dynpro ABAP, 154
- WHERE clause, 93, 99
- Where-used list, 48
- WRITE_ACCESS, 134
- Write lock, 186

**V**
- Value attribute, 24, 54, 146
- Vertical mapping, 44, 46, 47
- Visibility, 24, 57, 94
  - Private, 24, 38, 163
  - Protected, 24, 38, 119, 149
  - Public, 24, 143, 149, 163