11 Creating Packages

☑Identify the benefits and the components of packages

What is package?: 10.1

Reasons to use packages: 10.2

What is a package?

- A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions
- A package is compiled and stored in the database, where many applications can share its contents
- A package always has a **specification**, which declares the **public items** that can be referenced from outside the package
- If the public items include cursors or subprograms, then the package must also have a **body**
    - The body must define queries for public cursors and code for public subprograms
    - The body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package
    - Finally, the body can have an **initialization part**, whose statements initialize variables and do other one-time setup steps, and an exception-handling part
    - You can change the body without changing the specification or the references to the public items; therefore, you can think of the package body as a black box
- In either the package specification or package body, you can map a package subprogram to an external Java or C subprogram by using a call specification, which maps the external subprogram name, parameter types, and return type to their SQL counterparts
- The **AUTHID clause** of the package specification determines whether the subprograms and cursors in the package run with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker
- The **ACCESSIBLE BY clause** of the package specification lets you specify a white list of PL/SQL units that can access the package. You use this clause in situations like these:
    - You implement a PL/SQL application as several packages—one package that provides the application programming interface (API) and helper packages to do the work
        - You want clients to have access to the API, but not to the helper packages. Therefore, you omit the `ACCESSIBLE BY` clause from the API package specification and include it in each helper package specification, where you specify that only the API package can access the helper package
    - You create a utility package to provide services to some, but not all, PL/SQL units in the same schema
        - To restrict use of the package to the intended units, you list them in the `ACCESSIBLE BY` clause in the package specification

Reasons to use packages

Packages support the development and maintenance of reliable, reusable code with the following features:

- Modularity

    - Packages let you encapsulate logically related types, variables, constants, subprograms, cursors, and exceptions in named PL/SQL modules
    - You can make each package easy to understand, and make the interfaces between packages simple, clear, and well defined
    - This practice aids application development
- Easier Application Design
    - When designing an application, all you need initially is the interface information in the package specifications
    - You can code and compile specifications without their bodies
    - Next, you can compile standalone subprograms that reference the packages
    - You need not fully define the package bodies until you are ready to complete the application
- Hidden Implementation Details
    - Packages let you share your interface information in the package specification, and hide the implementation details in the package body
    - Hiding the implementation details in the body has these advantages:
        - You can change the implementation details without affecting the application interface
        - Application users cannot develop code that depends on implementation details that you might want to change
- Added Functionality
    - Package public variables and cursors can persist for the life of a session
    - They can be shared by all subprograms that run in the environment
    - They let you maintain data across transactions without storing it in the database
    - (Note: package public variables and cursors do not persist for the session in all situations, refer to topic "Package State")
- Better Performance
    - The first time you invoke a package subprogram, Oracle Database loads the whole package into memory. Subsequent invocations of other subprograms in same the package require no disk I/O
    - Packages prevent cascading dependencies and unnecessary recompiling. For example, if you change the body of a package function, Oracle Database does not recompile other subprograms that invoke the function, because these subprograms depend only on the parameters and return value that are declared in the specification
- Easier to Grant Roles
    - You can grant roles on the package, instead of granting roles on each object in the package

> Note: You cannot reference host variables from inside a package

✅Work with packages (create package specification and body, invoke package subprograms, remove a package and display package information)

Package specification: 10.3

Package body: 10.4

Instantiation and initialization: 10.5

Package state: 10.6

Serially reusable: 10.7

Writing guidelines: 10.8

CREATE PACKAGE

CREATE PACKAGE BODY

ALTER PACKAGE

DROP PACKAGE

Package specification

- A **package specification** declares **public items**
- The scope of a public item is the schema of the package
- A public item is visible everywhere in the schema
- To reference a public item that is in scope but not visible, qualify it with the package name
- To restrict the use of your package to specified PL/SQL units, include the `ACCESSIBLE BY` clause in the package specification

Appropriate public items

- Types, variables, constants, subprograms, cursors, and exceptions used by multiple subprograms
  - A type defined in a package specification is either a PL/SQL user-defined subtype or a PL/SQL composite type
  - A PL/SQL composite type defined in a package specification is incompatible with an identically defined local or standalone type
- Associative array types of standalone subprogram parameters
  - You cannot declare an associative array type at schema level. Therefore, to pass an associative array variable as a parameter to a standalone subprogram, you must declare the type of that variable in a package specification
  - Doing so makes the type available to both the invoked subprogram (which declares a formal parameter of that type) and to the invoking subprogram or anonymous block (which declares a variable of that type)
- Variables that must remain available between subprogram invocations in the same session
- Subprograms that read and write public variables ("get" and "set" subprograms)
  - Provide these subprograms to discourage package users from reading and writing public variables directly
- Subprograms that invoke each other
  - You need not worry about compilation order for package subprograms, as you must for standalone subprograms that invoke each other
- Overloaded subprograms
  - Overloaded subprograms are variations of the same subprogram. That is, they have the same name but different formal parameters

> Note: You cannot reference remote package public variables, even indirectly. For example, if a subprogram refers to a package public variable, you cannot invoke the subprogram through a database link

Creating package specifications: CREATE PACKAGE statement

- The `CREATE PACKAGE` statement creates or replaces the specification for a stored package, which is an encapsulated collection of related procedures, functions, and other program objects stored as a unit in the database
- The **package specification** declares these objects. The **package body**, specified subsequently, defines these objects
- Prerequisites
    - To create or replace a package in your schema, you must have the `CREATE PROCEDURE` system privilege
    - To create or replace a package in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege
    - To embed a `CREATE PACKAGE` statement inside an Oracle database precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language
- syntax:

```
CREATE [OR REPLACE] [EDITIONABLE | NONEDITIONABLE] PACKAGE
plsql_package_source
[schema.] package_name [sharing_clause]
[default_collation | invokers_rights | accessible_by] {IS | AS}
package_item_list END [package_name] ;
```

    - package_item_list:

```
type_definition | cursor_declaration | item_declaration |
package_function_declaration | package_procedure_declaration
```

        - Defines every type in the package and declares every cursor and subprogram in the package
        - Except for polymorphic table functions, every declaration must have a corresponding definition in the package body
        - The headings of corresponding declarations and definitions must match word for word, except for white space
        - Package polymorphic table function must be declared in the same package as their implementation package
        - `PRAGMA AUTONOMOUS_TRANSACTION` cannot appear here
    - package_function_declaration
        - similar to nested function declaration, except for properties: only accessible_by, deterministic, pipelined, parallel_enable, result_cache are allowed
    - package_procedure_declaration

        - similar to nested procedure declaration, except only accessible_by is allowed as a subprogram property

    - OR REPLACE

        - Re-creates the package if it exists, and recompiles it
        - Users who were granted privileges on the package before it was redefined can still access the package without being regranted the privileges
        - If any function-based indexes depend on the package, then the database marks the indexes `DISABLED`

    - [EDITIONABLE | NONEDITIONABLE]

        - Specifies whether the package is an editioned or noneditioned object if editioning is enabled for the schema object type `PACKAGE` in schema. Default: `EDITIONABLE`

Package body

- If a package specification declares cursors or subprograms, then a package body is required; otherwise, it is optional
- The package body and package specification must be in the same schema
- Every cursor or subprogram declaration in the package specification must have a corresponding definition in the package body
    - The headings of corresponding subprogram declarations and definitions must match word for word, except for white space
- The cursors and subprograms declared in the package specification and defined in the package body are **public items** that can be referenced from outside the package
- The package body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package
- Finally, the body can have an **initialization part**, whose statements initialize public variables and do other one-time setup steps
    - The initialization part runs only the first time the package is referenced
    - The initialization part can include an exception handler
- You can change the package body without changing the specification or the references to the public items

CREATE PACKAGE BODY statement

- The `CREATE PACKAGE BODY` statement creates or replaces the body of a stored package, which is an encapsulated collection of related procedures, stored functions, and other program objects stored as a unit in the database
- The **package body** defines these objects
- The **package specification**, defined in an earlier `CREATE PACKAGE` statement, declares these objects
- Packages are an alternative to creating procedures and functions as standalone schema objects
- Prerequisites

    - To create or replace a package in your schema, you must have the `CREATE PROCEDURE` system privilege
    - To create or replace a package in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege
    - In both cases, the package body must be created in the same schema as the package
    - To embed a `CREATE PACKAGE BODY` statement inside an the database precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language

- syntax:

```
CREATE [OR REPLACE] [EDITIONABLE | NONEDITIONABLE] PACKAGE BODY package_name
{IS | AS} declare_section [initialize_section] END [package_name] ;
```

    - initialize_section

```
BEGIN statement [EXCEPTION exception_handler]
```

        - Initializes variables and does any other one-time setup steps

    - OR REPLACE

        - Re-creates the package body if it exists, and recompiles it
        - Users who were granted privileges on the package body before it was redefined can still access the package without being regranted the privileges

    - [EDITIONABLE | NONEDITIONABLE]

        - If you do not specify this property, then the package body inherits `EDITIONABLE` or `NONEDITIONABLE` from the package specification
        - If you do specify this property, then it must match that of the package specification

    - declare_section

        - Has a definition for every cursor and subprogram declaration in the package specification
        - The headings of corresponding subprogram declarations and definitions must match word for word, except for white space
        - Can also declare and define private items that can be referenced only from inside the package
        - `PRAGMA AUTONOMOUS_TRANSACTION` cannot appear here

Package instantiation and initialization

- When a session references a package item, Oracle Database instantiates the package for that session
- Every session that references a package has its own instantiation of that package
- When Oracle Database instantiates a package, it initializes it
- Initialization includes whichever of the following are applicable:

    - Assigning initial values to public constants
    - Assigning initial values to public variables whose declarations specify them
    - Executing the initialization part of the package body

Package state

- The values of the variables, constants, and cursors that a package declares (in either its specification or body) comprise its **package state**
- If a PL/SQL package declares at least one variable, constant, or cursor, then the package is **stateful**; otherwise, it is **stateless**
- Each session that references a package item has its own instantiation of that package. If the package is stateful, the instantiation includes its state
- The package state persists for the life of a session, except in these situations:
    - The package is `SERIALLY_REUSABLE`
    - The package body is recompiled
        - If the body of an instantiated, stateful package is recompiled (either explicitly, with the `ALTER PACKAGE` statement, or implicitly), the next invocation of a subprogram in the package causes Oracle Database to discard the existing package state and raise the exception ORA-04068
        - After PL/SQL raises the exception, a reference to the package causes Oracle Database to re-instantiate the package, which re-initializes it. Therefore, previous changes to the package state are lost
    - Any of the session's instantiated packages are invalidated and revalidated
        - All of a session's package instantiations (including package states) can be lost if any of the session's instantiated packages are invalidated and revalidated
- Oracle Database treats a package as stateless if its state is constant for the life of a session (or longer). This is the case for a package whose items are all compile-time constants
- A **compile-time constant** is a constant whose value the PL/SQL compiler can determine at compilation time
    - A constant whose initial value is a literal is always a compile-time constant
    - A constant whose initial value is not a literal, but which the optimizer reduces to a literal, is also a compile-time constant
    - Whether the PL/SQL optimizer can reduce a nonliteral expression to a literal depends on optimization level. Therefore, a package that is stateless when compiled at one optimization level might be stateful when compiled at a different optimization level

SERIALLY_REUSABLE packages

- `SERIALLY_REUSABLE` packages let you design applications that manage memory better for scalability
- If a package is not `SERIALLY_REUSABLE`, its package state is stored in the user global area (UGA) for each user. Therefore, the amount of UGA memory needed increases linearly with the number of users, limiting scalability
- The package state can persist for the life of a session, locking UGA memory until the session ends. In some applications, such as Oracle Office, a typical session lasts several days
- If a package is `SERIALLY_REUSABLE`, its package state is stored in a work area in a small pool in the system global area (SGA)

- The package state persists only for the life of a server call. After the server call, the work area returns to the pool
- If a subsequent server call references the package, then Oracle Database reuses an instantiation from the pool. Reusing an instantiation re-initializes it; therefore, changes made to the package state in previous server calls are invisible
- > Note: Trying to access a `SERIALLY_REUSABLE` package from a database trigger, or from a PL/SQL subprogram invoked by a SQL statement, raises an error

Creating SERIALLY_REUSABLE packages

- To create a `SERIALLY_REUSABLE` package, include the `SERIALLY_REUSABLE` pragma in the package specification and, if it exists, the package body
- like so:

```
CREATE OR REPLACE PACKAGE mypack AS
  PRAGMA SERIALLY_REUSABLE;
  TYPE myrectype IS RECORD (n NUMBER);
  myrec myrectype;
  CURSOR mycur RETURN myrec%TYPE;
  i NUMBER;
END;
/
CREATE OR REPLACE PACKAGE BODY mypack AS
  PRAGMA SERIALLY_REUSABLE;
  CURSOR mycur RETURN myrec%TYPE IS SELECT id FROM emp;
BEGIN
  i := 10;
END;
/
```

SERIALLY_REUSABLE package work unit

- For a `SERIALLY_REUSABLE` package, the work unit is a server call
- You must use its public variables only within the work unit
- > Note: If you make a mistake and depend on the value of a public variable that was set in a previous work unit, then your program can fail. PL/SQL cannot check for such cases
- After the work unit (server call) of a SERIALLY_REUSABLE package completes, Oracle Database does the following:
    - Closes any open cursors
    - Frees some nonreusable memory (for example, memory for collection and long VARCHAR2 variables)
    - Returns the package instantiation to the pool of reusable instantiations kept for this package

- Example (basically, variables reset everytime to original values after server call in serially reusable packages):

```
CREATE OR REPLACE PACKAGE pack IS
  n number := 5;
END pack;
/
CREATE OR REPLACE PACKAGE srpack IS
  PRAGMA SERIALLY_REUSABLE;
  n number := 5;
END srpack;
/
BEGIN
  pack.n := 99;
  srpack.n := 99;
END;
/
BEGIN
  DBMS_OUTPUT.PUT_LINE('pack.n: '||pack.n);
  DBMS_OUTPUT.PUT_LINE('srpack.n: '||srpack.n);
END;
/
-- result:
pack.n: 99
srpack.n: 5
```

Explicit cursors in SERIALLY_REUSABLE packages

- An explicit cursor in a `SERIALLY_REUSABLE` package remains open until either you close it or its work unit (server call) ends
- To re-open the cursor, you must make a new server call. A server call can be different from a subprogram invocation
- In contrast, an explicit cursor in a package that is not `SERIALLY_REUSABLE` remains open until you either close it or disconnect from the session
    - Fetching from this cursor would also use the state, i.e. fetching the next time would fetch the following row (%rowcount + 1), as long as it is within the same session

Package writing guidelines

- Become familiar with the packages that Oracle Database supplies, and avoid writing packages that duplicate their features
- Keep your packages general so that future applications can reuse them
- Design and define the package specifications before the package bodies
- In package specifications, declare only items that must be visible to invoking programs
    - This practice prevents other developers from building unsafe dependencies on your implementation details and reduces the need for recompilation
    - If you change the package specification, you must recompile any subprograms that invoke the public subprograms of the package. If you change only the package body, you need not

recompile those subprograms

- Declare public cursors in package specifications and define them in package bodies
  - This practice lets you hide cursors' queries from package users and change them without changing cursor declarations
- Assign initial values in the initialization part of the package body instead of in declarations. This practice has these advantages:
  - The code for computing the initial values can be more complex and better documented
  - If computing an initial value raises an exception, the initialization part can handle it with its own exception handler
- If you implement a database application as several PL/SQL packages—one package that provides the API and helper packages to do the work, then make the helper packages available only to the API package

ALTER PACKAGE statement

- The `ALTER PACKAGE` statement explicitly recompiles a package specification, body, or both
- Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead
- Because all objects in a package are stored as a unit, the `ALTER PACKAGE` statement recompiles all package objects
- You cannot use the `ALTER PROCEDURE` statement or `ALTER FUNCTION` statement to recompile individually a procedure or function that is part of a package
- > Note: This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the `CREATE PACKAGE` statement, or the `CREATE PACKAGE BODY` statement with the `OR REPLACE` clause
- Prerequisites
  - If the package is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the package must be in your schema or you must have `ALTER ANY PROCEDURE` system privilege
- syntax:

```
ALTER PACKAGE [schema.] package_name {package_compile_clause |
[EDITIONABLE | NONEDTIONABLE]}
```

  - package_compile_clause:

```
COMPILE [DEBUG] [PACKAGE | SPECIFICATION | BODY] [compiler_parameters]
[REUSE SETTINGS]
```

    - Recompiles the package specification, body, or both

DROP PACKAGE statement

- The `DROP PACKAGE` statement drops a stored package from the database
- This statement drops the body and specification of a package
- > Note: Do not use this statement to drop a single object from a package. Instead, re-create the package without the object using the `CREATE PACKAGE` statement and `CREATE PACKAGE BODY` statement with the `OR REPLACE` clause
- Prerequisites
  - The package must be in your schema or you must have the `DROP ANY PROCEDURE` system privilege
- syntax:

```
DROP PACKAGE [BODY] [schema.] package ;
```

  - BODY:
    - Drops only the body of the package
    - If you omit this clause, then the database drops both the body and specification of the package
    - When you drop only the body of a package but not its specification, the database does not invalidate dependent objects
    - However, you cannot invoke a procedure or stored function declared in the package specification until you re-create the package body
  - package
    - Name of the package to be dropped
    - The database invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped package
    - If any statistics types are associated with the package, then the database disassociates the statistics types with the `FORCE` clause and drops any user-defined statistics collected with the statistics types

☑Overload package subprograms and use forward declarations

Forward declaration: 8.6

Overloaded subprograms: 8.9

Package subprogram overloading

See chapter 8 8_subprograms.md#overloading-subprograms

Forward declaration in packages

See chapter 9 9_procedures.md#forward-declaration