20 Choosing a Programming Environment

To choose a programming environment for a development project, read:

- The topics in this chapter and the documents to which they refer.
- The platform-specific documents that explain which compilers and development tools your platforms support.

Sometimes the choice of programming environment is obvious, for example:

- Pro*COBOL does not support ADTs or collection types, while Pro*C/C++ does.

If no programming language provides all the features you need, you can use multiple programming languages, because:

- Every programming language in this chapter can invoke PL/SQL and Java stored subprograms. (Stored subprograms include triggers and ADT methods.)
- PL/SQL, Java, SQL, and Oracle Call Interface (OCI) can invoke external C subprograms.
- External C subprograms can access Oracle Database using SQL, OCI, or Pro*C (but not C++).

Topics:

See Also:

Developing Applications with Multiple Programming Languages for more information about multilanguage programming

20.1 Overview of Application Architecture

In this topic, **application architecture** refers to the computing environment in which a database application connects to an Oracle Database.

Topics:

- Client/Server Architecture
- Server-Side Programming
- Two-Tier and Three-Tier Architecture

See Also:

Oracle Database Concepts for more information about application architecture

20.1.1 Client/Server Architecture

In a traditional client/server program, your application code runs on a client system; that is, a system other than the database server. Database calls are transmitted from the client system to the database server. Data is transmitted from the client to the server for insert and update operations and returned from the server to the client for query operations. The data is processed on the client system. Client/server programs are typically written by using precompilers, whereas SQL statements are embedded within the code of another language such as C, C++, or COBOL.

See Also:

Oracle Database Concepts for more information about client/server architecture

20.1.2 Server-Side Programming

You can develop application logic that resides entirely inside the database by using triggers that are executed automatically when changes occur in the database or stored subprograms that are invoked explicitly. Off-loading the work from your application lets you reuse code that performs verification and cleanup and control database operations from a variety of clients. For example, by making stored subprograms invocable through a web server, you can construct a web-based user interface that performs the same functions as a client/server application.

See Also:

Oracle Database Concepts for more information about server-side programming

20.1.3 Two-Tier and Three-Tier Architecture

Client/server computing is often referred to as a **two-tier model**: your application communicates directly with the database server. In the **three-tier model**, a separate application server processes the requests. The application server might be a basic web server, or might perform advanced functions like caching and load-balancing. Increasing the processing power of this middle tier lets you lessen the resources needed by client systems, resulting in a **thin client configuration** in which the client system might need only a web browser or other means of sending requests over the TCP/IP or HTTP protocols.

See Also:

Oracle Database Concepts for more information about multitier architecture

20.2 Overview of the Program Interface

The **program interface** is the software layer between a database application and Oracle Database. The program interface:

- Provides a security barrier, preventing destructive access to the SGA by client user processes
- Acts as a communication mechanism, formatting information requests, passing data, and trapping and returning errors
- Converts and translates data, particularly between different types of computers or to external user program data types

The Oracle code acts as a server, performing database tasks on behalf of an application (a client), such as fetching rows from data blocks. The program interface consists of several parts, provided by both Oracle Database software and operating system-specific software.

See Also:

Oracle Database Concepts for more information about the program interface

Topics:

- User Interface
- Stateful and Stateless User Interfaces

20.2.1 User Interface

The **user interface** is what your application displays to end users. It depends on the technology behind the application and the needs of the users themselves. Experienced users can enter SQL statements that are passed on to the database. Novice users can be shown a graphical user interface that uses the graphics libraries of the client system (such as Windows or X-Windows). Any traditional user interface can also be provided in a web browser through HTML and Java.

20.2.2 Stateful and Stateless User Interfaces

In traditional client/server applications, the application can keep a record of user actions and use this information over the course of one or more sessions. For example, past choices can be presented in a menu so that they not be entered again. When the application can save information in this way, the application is considered **stateful**.

Web or thin-client applications that are **stateless** are easier to develop. Stateless applications gather all the required information, process it using the database, and then start over with the next user. This is a popular way to process single-screen requests such as customer registration.

There are many ways to add stateful action to web applications that are stateless by default. For example, an entry form on one web page can pass information to subsequent web pages, enabling you to construct a wizard-like interface that remembers user choices through several different steps. You can use cookies to store small items of information about the client system, and retrieve them when the user returns to a website. You can use servlets to keep a database session open and store variables between requests from the same client.

20.3 Overview of PL/SQL

PL/SQL, the Oracle procedural extension of SQL, is a completely portable, high-performance transaction-processing language. PL/SQL lets you manipulate data with SQL statements; control program flow with conditional selection and loops; declare constants and variables; define subprograms; define types, subtypes, and ADTs and declare variables of those types; and trap runtime errors.

Applications written in any Oracle Database programmatic interface can invoke PL/SQL stored subprograms and send blocks of PL/SQL code to Oracle Database for execution. Third-generation language (3GL) applications can access PL/SQL scalar and composite data types through host variables and implicit data type conversion. A 3GL language is easier than assembler language for a human to understand and includes features such as named variables. Unlike a fourth-generation language (4GL), it is not specific to an application domain.

You can use PL/SQL to develop stored procedures that can be invoked by a web client.

See Also:

- *Oracle Database PL/SQL Language Reference* for information about the advantages, main features, and architecture of PL/SQL

20.4 Overview of Oracle Database Java Support

This section provides an overview of Oracle Database features that support Java applications. The database includes the core JDK libraries such as `java.lang`, `java.io`, and so on. The database supports client-side Java standards such as JDBC and SQLJ, and provides server-side JDBC driver that enables data-intensive Java code to run within the database.

Topics:

- Overview of Oracle JVM

- Overview of Oracle JDBC
- Overview of Oracle SQLJ
- Comparison of Oracle JDBC and Oracle SQLJ
- Overview of Java Stored Subprograms
- Overview of Oracle Database Web Services

See Also:

- *Oracle Database Java Developer's Guide*
- *Oracle Database JDBC Developer's Guide*

20.4.1 Overview of Oracle JVM

Oracle JVM, the Java Virtual Machine provided with the Oracle Database, is compliant with the J2SE version 1.5.x specification and supports the database session architecture.

Any database session can activate a dedicated JVM. All sessions share the same JVM code and statics; however, private states for any given session are held, and subsequently garbage collected, in an individual session space.

This design provides these benefits:

- Java applications have the same session isolation and data integrity as SQL operations.
- You need not run Java in a separate process for data integrity.
- Oracle JVM is a robust JVM with a small memory footprint.
- The JVM has the same linear Symmetric Multiprocessing (SMP) scalability as the database and can support thousands of concurrent Java sessions.

Oracle JVM works consistently with every platform supported by Oracle Database. Java applications that you develop with Oracle JVM can easily be ported to any supported platform.

Oracle JVM includes a deployment-time native compiler that enables Java code to be compiled once, stored in executable form, shared among users, and invoked more quickly and efficiently.

Security features of the database are also available with Oracle JVM. Java classes must be loaded in a database schema (by using Oracle JDeveloper, a third-party IDE, SQL*Plus, or the `loadjava` utility) before they can be called. Java class calls are secured and controlled through database authentication and authorization, Java 2 security, and invoker's rights (IR) or definer's rights (DR).

Effective with Oracle Database 12*c* Release 1 (12.1.0.1), Oracle JVM provides complete support for the latest Java Standard Edition. Compatibility with latest Java standards increases application portability and enables direct execution of client-side Java classes in the database.

See Also:

- *Oracle Database Concepts* for additional general information about Oracle JVM
- *Oracle Database Java Developer's Guide* for information about support for the latest Java Standard Edition

20.4.2 Overview of Oracle JDBC

Java Database Connectivity (JDBC) is an Applications Programming Interface (API) that enables Java to send SQL statements to an object-relational database such as Oracle Database.

Oracle Database includes these extensions to the JDBC 1.22 standard:

- Support for Oracle data types
- Performance enhancement by row prefetching
- Performance enhancement by execution batching
- Specification of query column types to save round trips
- Control of `DatabaseMetaData` calls

Oracle Database supports all APIs from the JDBC 2.0 standard, including the core APIs, optional packages, and numerous extensions. Some highlights include datasources, JTA, and distributed transactions.

Oracle Database supports these features from the JDBC 3.0 standard:

- Support for JDK 1.5.
- Toggling between local and global transactions.
- Transaction savepoints.
- Reuse of prepared statements by connection pools.

Note:

JDBC code and SQLJ code interoperate.

Topics:

- Oracle JDBC Drivers
- Sample JDBC 2.0 Program
- Sample Pre-2.0 JDBC Program

See Also:

- *Oracle Database Concepts* for additional general information about Java support in Oracle Database
- Comparison of Oracle JDBC and Oracle SQLJ

20.4.2.1 Oracle JDBC Drivers

The JDBC standard defines four types of JDBC drivers:

Type	Description
1	A JDBC-ODBC bridge. Software must be installed on client systems.
2	Native methods (calls C or C++) and Java methods. Software must be installed on the client.
3	Pure Java. The client uses sockets to call middleware on the server.
4	The most pure Java solution. Talks directly to the database by using Java sockets.

JDBC is based on Part 3 of the SQL standard, "Call-Level Interface."

You can use JDBC to do dynamic SQL. In dynamic SQL, the embedded SQL statement to be executed is not known before the application is run and requires input to build the statement.

The drivers that are implemented by Oracle have extensions to the capabilities in the JDBC standard that was defined by Sun Microsystems.

Topics:

- JDBC Thin Driver
- JDBC OCI Driver
- JDBC Server-Side Internal Driver

See Also:

- *Oracle Database Concepts* for additional general information about JDBC drivers
- *Oracle Database JDBC Developer's Guide* for more information about JDBC

20.4.2.1.1 JDBC Thin Driver

The JDBC Thin driver is a Type 4 (100% pure Java) driver that uses Java sockets to connect directly to a database server. It has its own implementation of a Two-Task Common (TTC), a lightweight implementation of TCP/IP from Oracle Net. It is written entirely in Java and is therefore platform-independent.

The thin driver does not require Oracle software on the client side. It does need a TCP/IP listener on the server side. Use this driver in Java applets that are downloaded into a web browser or in applications for which you do not want to install Oracle client software. The thin driver is self-contained, but it opens a Java socket, and thus can run only in a browser that supports sockets.

20.4.2.1.2 JDBC OCI Driver

The JDBC OCI driver is a Type 2 JDBC driver. It makes calls to OCI written in C to interact with Oracle Database, thus using native and Java methods.

The OCI driver provides access to more features than the thin driver, such as Transparent Application Fail-Over, advanced security, and advanced LOB manipulation.

The OCI driver provides the highest compatibility between different Oracle Database versions. It also supports all installed Oracle Net adapters, including IPC, named pipes, TCP/IP, and IPX/SPX.

Because it uses native methods (a combination of Java and C) the OCI driver is platform-specific. It requires a client installation of version Oracle8*i* or later including Oracle Net, OCI libraries, CORE libraries, and all other dependent files. The OCI driver usually runs faster than the thin driver.

The OCI driver is not appropriate for Java applets, because it uses a C library that is platform-specific and cannot be downloaded into a web browser. It is usable in J2EE components running in middle-tier application servers, such as Oracle Application Server. Oracle Application Server provides middleware services and tools that support access between applications and browsers.

20.4.2.1.3 JDBC Server-Side Internal Driver

The JDBC server-side internal driver is a Type 2 driver that runs inside the database server, reducing the number of round trips needed to access large amounts of data. The driver, the Java server VM, the database, the Java native compiler (which speeds execution by as much as 10 times), and the SQL engine all run within the same address space.

This driver provides server-side support for any Java program used in the database. You can also call PL/SQL stored subprograms and triggers.

The server driver fully supports the same features and extensions as the client-side drivers.

20.4.2.2 Sample JDBC 2.0 Program

This example shows the recommended technique for looking up a data source using JNDI in JDBC 2.0:

```
// import the JDBC packages
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.pool.*;
...
   InitialContext ictx = new InitialContext();
   DataSource ds = (DataSource)ictx.lookup("jdbc/OracleDS");
   Connection conn = ds.getConnection();
   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT last_name FROM employees");
   while ( rs.next() ) {
   out.println( rs.getString("ename") + "<br>");
   }
conn.close();
```

20.4.2.3 Sample Pre-2.0 JDBC Program

This source code registers an Oracle JDBC thin driver, connects to the database, creates a `Statement` object, runs a query, and processes the result set.

The `SELECT` statement retrieves and lists the contents of the `last_name` column of the `hr.employees` table.

```
import java.sql.*
import java.math.*
import java.io.*
import java.awt.*

class JdbcTest {
  public static void main (String args []) throws SQLException {
    // Load Oracle driver
    DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

    // Connect to the local database
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:thin:@myhost:1521:orcl",
                                   "hr", "password");

    // Query the employee names
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("SELECT last_name FROM employees");

    // Print the name out
    while (rset.next ())
      System.out.println (rset.getString (1));
    // Close the result set, statement, and the connection
    rset.close();
    stmt.close();
    conn.close();
  }
}
```

One Oracle Database extension to the JDBC drivers is a form of the `getConnection()` method that uses a `Properties` object. The `Properties` object lets you specify user, password, database information, row prefetching, and execution batching.

To use the OCI driver in this code, replace the `Connection` statement with this code, where `MyHostString` is an entry in the `tnsnames.ora` file:

```
Connection conn = DriverManager.getConnection ("jdbc:oracle:oci8:@MyHostString",
    "hr", "password");
```

If you are creating an applet, then the `getConnection()` and `registerDriver()` strings are different.

20.4.3 Overview of Oracle SQLJ

Note:

In this guide, **SQLJ** refers to Oracle SQLJ and its extensions.

SQLJ is an ANSI SQL-1999 standard for embedding SQL statements in Java source code. SQLJ provides a simpler alternative to JDBC for client-side SQL data access from Java.

A SQLJ source file contains Java source with embedded SQL statements. Oracle SQLJ supports dynamic and static SQL. Support for dynamic SQL is an Oracle extension to the SQLJ standard.

The Oracle SQLJ translator performs these tasks:

- Translates SQLJ source to Java code with calls to the SQLJ runtime driver. The SQLJ translator converts the source code to pure Java source code and can check the syntax and semantics of static SQL statements against a database schema and verify the type compatibility of host variables with SQL types.
- Compiles the generated Java code with the Java compiler.
- (Optional) Creates profiles for the target database. SQLJ generates "profile" files with customization specific to Oracle Database.

SQLJ is integrated with JDeveloper. Source-level debugging support for SQLJ is available in JDeveloper.

This is an example of a simple SQLJ executable statement, which returns one value because `employee_id` is unique in the `employee` table:

```
String name;
#sql  { SELECT first_name INTO :name FROM employees WHERE employee_id=112 };
System.out.println("Name is " + name + ", employee number = " + employee_id);
```

Each host variable (or qualified name or complex Java host expression) included in a SQL expression is preceded by a colon (:). Other SQLJ statements declare Java types. For example, you can declare an iterator (a construct related to a database cursor) for queries that retrieve many values, as follows:

```
#sql iterator EmpIter (String EmpNam, int EmpNumb);
```

See Also:

Oracle Database SQLJ Developer's Guide for more examples and details about Oracle SQLJ syntax

Topics:

- Benefits of SQLJ

See Also:

Oracle Database Concepts for additional general information about SQLJ

20.4.3.1 Benefits of SQLJ

Oracle SQLJ extensions to Java enable rapid development and easy maintenance of applications that perform database operations through embedded SQL.

In particular, Oracle SQLJ does this:

- Provides a concise, legible mechanism for database access from static SQL. Most SQL in applications is static. SQLJ provides more concise and less error-prone static SQL constructs than JDBC does.
- Provides an SQL Checker module for verification of syntax and semantics at translate time.
- Provides flexible deployment configurations, which makes it possible to implement SQLJ on the client, or middle tier.
- Supports a software standard. SQLJ is an effort of a group of vendors and is supported by all of them. Applications can access multiple database vendors.
- Provides source code portability. Executables can be used with all of the vendor DBMSs if the code does not rely on vendor-specific features.
- Enforces a uniform programming style for the clients and the servers.
- Integrates the SQLJ translator with **Oracle JDeveloper**, a graphical IDE that provides SQLJ translation, Java compilation, profile customizing, and debugging at the source code level, all in one step.
- Includes Oracle Database type extensions.

20.4.4 Comparison of Oracle JDBC and Oracle SQLJ

JDBC code and SQLJ code interoperate, enabling dynamic SQL statements in JDBC to be used with both static and dynamic SQL statements in SQLJ. A SQLJ iterator class corresponds to the JDBC result set.

Some differences between JDBC and SQLJ are:

- JDBC provides a complete dynamic SQL interface from Java to databases. It gives developers full control over database operations. SQLJ simplifies Java database programming to improve development productivity.
- JDBC provides fine-grained control of the execution of dynamic SQL from Java, whereas SQLJ provides a higher-level binding to SQL operations in a specific database schema.
- SQLJ source code is more concise than equivalent JDBC source code.
- SQLJ uses database connections to type-check static SQL code. JDBC, being a completely dynamic API, does not.
- SQLJ provides strong typing of query outputs and return parameters and provides type-checking on calls. JDBC passes values to and from SQL without compile-time type checking.
- SQLJ programs enable direct embedding of Java bind expressions within SQL statements. JDBC requires a separate get or set statement for each bind variable and specifies the binding by position number.
- SQLJ provides simplified rules for calling SQL stored subprograms.

  For example, the following four examples show, on successive lines, how to call a stored procedure or a stored function using either JDBC escape syntax or Oracle JDBC syntax:

  ```
  prepStmt.prepareCall("{call fun(?,?)}");       //stored proc. JDBC esc.
  prepStmt.prepareCall("{? = call fun(?,?)}");   //stored func. JDBC esc.
  prepStmt.prepareCall("begin fun(:1,:2);end;"); //stored proc. Oracle
  prepStmt.prepareCall("begin :1 := fun(:2,:3);end;"); //stored func. Oracle
  ```

  The SQLJ equivalent is:

```
#sql {call fun(param_list) };  //Stored procedure
// Declare x
...
#sql x = {VALUES(fun(param_list)) };  // Stored function
// where VALUES is the SQL construct
```

These benefits are common to SQLJ and JDBC:

- SQLJ source files can contain JDBC calls. SQLJ and JDBC are interoperable.
- PL/SQL and Java stored subprograms can be used interchangeably.

20.4.5 Overview of Java Stored Subprograms

Java stored subprograms enable you to implement programs that run in the database server and are independent of programs that run in the middle tier. Structuring applications in this way reduces complexity and increases reuse, security, performance, and scalability.

For example, you can create a Java stored subprogram that performs operations that require data persistence and a separate program to perform presentation or business logic operations.

Java stored subprograms interface with SQL using an execution model similar to that of PL/SQL.

See Also:

- *Oracle Database Concepts* for additional general information about Java stored subprograms
- *Oracle Database Java Developer's Guide* for complete information about Java stored subprograms

20.4.6 Overview of Oracle Database Web Services

Web services represent a distributed computing paradigm for Java application development that is an alternative to earlier Java protocols such as JDBC, and which enable applications to interact through the XML and web protocols. For example, an electronics parts vendor can provide a web-based programmatic interface to its suppliers for inventory management. The vendor can invoke a web service as part of a program and automatically order stock based on the data returned.

The key technologies used in web services are:

- Web Services Description Language (WSDL), which is a standard format for creating an XML document. WSDL describes what a web service can do, where it resides, and how to invoke it. Specifically, it describes the operations and parameters, including parameter types, provided by a web service. Also, a WSDL document describes the location, the transport protocol, and the invocation style for the web service.
- Simple Object Access Protocol (SOAP) messaging, which is an XML-based message protocol used by web services. SOAP does not prescribe a specific transport mechanism such as HTTP, FTP, SMTP, or JMS; however, most web services accept messages that use HTTP or HTTPS.
- Universal Description, Discovery, and Integration (UDDI) business registry, which is a directory that lists web services on the internet. The UDDI registry is often compared to a

telephone directory, listing unique identifiers (white pages), business categories (yellow pages), and instructions for binding to a service protocol (green pages).

Web services can use a variety of techniques and protocols. For example:

- Dispatching can occur in a synchronous (typical) or asynchronous manner.
- You can invoke a web service in an RPC-style operation in which arguments are sent and a response returned, or in a message style such as a one-way SOAP document exchange.
- You can use different encoding rules: literal or encoded.

You can invoke a web service statically, when you might know everything about it beforehand, or dynamically, in which case you can discover its operations and transport endpoints while using it.

Oracle Database can function as either a web service provider or as a web service consumer. When used as a provider, the database enables sharing and disconnected access to stored subprograms, data, metadata, and other database resources such as the queuing and messaging systems.

As a web service provider, Oracle Database provides a disconnected and heterogeneous environment that:

- Exposes stored subprograms independently of the language in which the subprograms are written
- Exposes SQL Queries and XQuery

See Also:

Oracle Database Concepts for additional general information about Oracle Database as a web service provider

20.5 Overview of JavaScript

This section provides you with an overview of how you can run server-side procedural logic in JavaScript using Oracle Multilingual Engine (MLE).

Topics:

- Multilingual Engine Overview
- MLE Concepts
- Understanding MLE Execution Context and Runtime Isolation
- MLE Environment Overview
- JavaScript MLE Modules Overview
- JavaScript MLE Call Specification Overview
- Invoking JavaScript in the Database
- Invoking JavaScript Using MLE Modules
- Invoking JavaScript Using Dynamic MLE Execution
- Privileges for Working with JavaScript in MLE
- Other Supported MLE Features

This section is intended to be an introduction to how you can run JavaScript in Oracle Database. For a more comprehensive guide, see *Oracle Database JavaScript Developer's Guide*.

20.5.1 Multilingual Engine Overview

Oracle Database Multilingual Engine (MLE) enables Oracle Database users to run JavaScript in Oracle Database using stored procedures and dynamic code snippets.

MLE employs a smart database (SmartDB) approach, allowing application logic and data to coexist in the same database. Processing data within the database helps mitigate against problems arising out of handling heavy data volumes and transferring information between servers (database, middle tier, and client).

Using the SmartDB approach, Oracle Database MLE provides these benefits:

- Running the logic in the database removes unnecessary data transfer over the network and improves performance of data-intensive operations.
- Storing and running requirements such as business rules inside the database ensures that every application follows the rules. Hence, implementing security and compliance requirements is simplified.
- Storing of commonly used functions in a central place can help with code reuse while avoiding code replication.

If you have data-intensive applications that use up unnecessary resources in a three-tier architecture, you can move the processing logic from the middle tier to the database for faster throughput, better security, and seamless data processing that happens closer to the database.

MLE introduces three kinds of schema objects: MLE modules, MLE environments, and MLE call specifications. MLE provides DDL extensions to create, alter, and drop these objects, and users with the right database privileges can use these extensions. These objects have dictionary views that you can query for information about the objects. MLE also enables you to perform post-execution debugging using the runtime states that are collected during the program runtime.

See Also:

Introduction to Multilingual Engine in *Oracle Database JavaScript Developer's Guide* to learn more about JavaScript and MLE.

20.5.2 MLE Concepts

MLE Module

An MLE module is a unit of JavaScript code that is stored in the database as a schema object.

MLE Function

An MLE function is exported by an MLE module and made available for calling from PL/SQL and SQL, as a function or procedure.

MLE Call Specification

An MLE call specification publishes the MLE functions (as PL/SQL functions or procedures) that you can call from SQL and PL/SQL.

MLE Execution Context

An MLE execution context encapsulates all runtime state associated with the execution of JavaScript code.

MLE Environment

An MLE environment configures the properties of MLE execution contexts.

Dynamic MLE

Dynamic MLE uses the `DBMS_MLE` PL/SQL package to enable direct execution of anonymous JavaScript code snippets.

20.5.3 Understanding MLE Execution Context and Runtime Isolation

An MLE execution context is a standalone, isolated runtime environment for JavaScript and encapsulates all runtime state associated with the execution of the JavaScript code in a session.

MLE uses execution contexts to enforce runtime state isolation between MLE modules, anonymous code snippets, and database users. Isolation of different applications in separate contexts within the same session prevents any interference between the applications. Runtime state isolation is important for languages such as JavaScript, where non-local variables have global scope but there is no control over their visibility.

MLE uses execution contexts to ensure that:

- Any code executing in one execution context cannot see or modify runtime state in another execution context.
- All code that shares an execution context has full access to all the runtime states in that context, for example, access to any predefined global variables.
- Different database users cannot observe each other's runtime states.
- All JavaScript code is evaluated using the user, roles, and schema that are in effect at the time of context creation.
- Each context is bound to a single session.

MLE uses execution contexts in two different scenarios:

- With call specifications, where implicit execution contexts are used for SQL and PL/SQL calls to MLE functions
- With dynamic MLE execution, where developers create and use dynamic MLE contexts explicitly

An MLE context has a lifetime that is limited to the session in which it was created. When a session ends or is reset, all the contexts created in that session are dropped. A session can run the JavaScript code as a dynamic MLE snippet or as a function that is exported from an MLE module.

When using multiple contexts, it is recommended that applications create necessary contexts only, and drop contexts when they are not referenced.

See Also:

About MLE Execution Contexts in *Oracle Database JavaScript Developer's Guide* for more information about MLE execution contexts.

20.5.4 MLE Environment Overview

You can use an MLE environment to configure specific properties of the execution contexts that MLE uses to run JavaScript code in dynamic MLE snippets and MLE call specifications.

With MLE environments, you can:

- Set JavaScript-specific language options for an execution context at runtime.
- Enable specific MLE modules that you want to import in an execution context. Environments enable developers to create mappings between module imports (`import * from <module-import>`) and modules as stored in the database. For instance, you can create an MLE environment, with "`a`" mapped to "`module_a`", as in `create mle env myEnv imports('a' module module_a);`.

MLE environments are schema objects that you can manage (create, modify, or drop) and reuse across multiple execution contexts. To create an environment, use the `CREATE MLE ENV` DDL statement. Alternatively, you can create an environment as an independent copy of an existing environment.

Note:

Any change made to the original environment after the cloning, is not propagated to the cloned environment.

Users must have the `CREATE MLE` privilege to create MLE modules and environments in their own schema, or the `CREATE ANY MLE` privilege to create MLE modules and environments in arbitrary schemas.

If no environment is explicitly specified, MLE uses a default environment (from the `SYS` schema) that configures default JavaScript-specific language options and makes built-in MLE modules available for import. The default environment is used for dynamic MLE contexts and module contexts.

Information about MLE environments is available in these dictionary views: `[user | all | dba | cdb]_mle_envs`.

See Also:

- Creating MLE Environments in the Database in *JavaScript Developer's Guide* for more information about creating MLE environments
- CREATE MLE ENV in *Oracle Database SQL Language Reference* for more information about the DDL commands used for MLE environments
- USER_MLE_ENVS, ALL_MLE_ENVS, DBA_MLE_ENVS, and CDB_MLE_ENVS in *Oracle Database Reference* for more information about the environment views

20.5.5 JavaScript MLE Modules Overview

You can have JavaScript code stored persistently as an MLE module in the database. An MLE module is a unit of MLE language code that is stored in the database as a schema object and contains code that is written in a single MLE language (in this case, JavaScript). An MLE module enables you to export MLE functions, which you can call from PL/SQL and SQL as a function or procedure.

MLE supports user-defined MLE modules and built-in MLE modules. Built-in modules are not deployed to the database like user-defined MLE modules, but are included as a part of MLE runtime. You cannot change or modify any Oracle-provided JavaScript module.

You can use DDL statements to manage (create, alter, drop) MLE modules as database objects. To get information about MLE modules and other schema objects, use the `USER_MLE_MODULES`, `ALL_MLE_MODULES`, `DBA_MLE_MODULES`, and `CDB_MLE_MODULES` dictionary views.

See Also:

- Using JavaScript Modules in MLE in *JavaScript Developer's Guide* for more information about MLE modules
- Managing JavaScript MLE Modules
- USER_MLE_MODULES, ALL_MLE_MODULES, DBA_MLE_MODULES, and CDB_MLE_MODULES in *Oracle Database Reference* for more information about the module views

20.5.6 JavaScript MLE Call Specification Overview

Call specifications publish JavaScript MLE functions so that they can be called from SQL and PL/SQL. When executing call specifications in a session, MLE loads the module specified in the call specification and calls the function(s) exported by that module.

You can create MLE call specifications using the `CREATE FUNCTION` or `CREATE PROCEDURE` DDL statement syntax with MLE-specific elements.

You can drop a call specification using the `DROP FUNCTION` or `DROP PROCEDURE` DDL statement.

You must have the `CREATE PROCEDURE` privilege to create MLE call specifications in your schema, and the `CREATE ANY PROCEDURE` privilege to create MLE call specifications in an arbitrary schema. You must have the `EXECUTE` privilege on an MLE call specification to call it.

The `MLE MODULE` clause in the `CREATE FUNCTION | PROCEDURE` statement specifies the MLE module that exports the underlying JavaScript function for the call specification. The specified module must always be in the same schema as the call specification being created.

MLE provides three dictionary views that you can query for information about MLE call specifications: `USER_MLE_PROCEDURES`, `ALL_MLE_PROCEDURES`, `DBA_MLE_PROCEDURES`, and `CDB_MLE_PROCEDURES`.

See Also:

- Creating an MLE Call Specification in *Oracle Database JavaScript Developer's Guide*
- CREATE PROCEDURE in *Oracle Database SQL Language Reference* for more information about the DDL commands used for MLE call specifications
- USER_MLE_PROCEDURES, ALL_MLE_PROCEDURES, DBA_MLE_PROCEDURES, and CDB_MLE_PROCEDURES in *Oracle Database Reference* for more information about the environment views

20.5.7 Invoking JavaScript in the Database

You can invoke JavaScript in either of the following ways:

- Using MLE module calls; whereby PL/SQL code references the functions that are exported in JavaScript modules
- Using the `DBMS_MLE` PL/SQL package in Dynamic MLE; whereby you can run JavaScript code snippets using the procedures defined in the package.

See Also:

- Invoking JavaScript Using MLE Modules
- Invoking JavaScript Using Dynamic MLE Execution

20.5.8 Invoking JavaScript Using MLE Modules

This section provides a brief background of MLE modules and how you can use them to invoke JavaScript in the database.

For the complete documentation related to MLE modules, see *Oracle Database JavaScript Developer's Guide*.

Topics:

- Using MLE Module Contexts
- Specifying an Environment for Call Specifications
- Managing JavaScript MLE Modules
- Running JavaScript Code Using MLE Modules

20.5.8.1 Using MLE Module Contexts

When running a call specification in a session, the JavaScript runtime engine loads the JavaScript module that has the exported functions (JavaScript functions) to be called in the call specification. In any given session, execution contexts for each module called in call specifications are created implicitly on demand. Calls from SQL and PL/SQL to MLE functions are run in a dedicated execution context for the MLE module and for the user on whose behalf the call is executed.

MLE uses execution contexts to maintain runtime state isolation. Call specifications are isolated into separate contexts when they do not share the same user, module, or environment.

Here are some important points to note while using execution contexts with MLE modules:

- Execution contexts separate the runtime state of different users and of different MLE modules, including separate execution contexts in cases where code from a same MLE module is executed on behalf of different database users.
- MLE creates dedicated execution context for each combination of MLE module and MLE environment. Thus, two call specifications that specify either different modules or different environments are executed in separate module contexts, preventing incompatible modules from interfering with each other.
- Within the same session, MLE may employ multiple module contexts to execute call specifications.
- The runtime representation of a module is stateful. A state constitutes elements such as variables in the JavaScript module itself and variables in the global scope that are accessible to the code in the module.

See Also:

About MLE Execution Contexts in *JavaScript Developer's Guide* for more information about MLE module execution contexts

20.5.8.2 Specifying an Environment for Call Specifications

An MLE call specification can optionally define the environment for the module context that executes the MLE call specification. To define the environment for a module context, include the `ENV` clause when creating an MLE call specification:

```
CREATE PROCEDURE scott.print_hello(name IN VARCHAR2) AS
 MLE MODULE scott."mymodule"
 ENV scott."myenv"
 SIGNATURE 'printHello';
```

All calls to the scott.print procedure are executed in a module context in which the `scott.mymodule` module is loaded. In addition, the module context is configured according to the `scott.myenv` environment.

Module contexts are separated by environment; if two call specifications refer to the same module but different environments, separate module contexts are created, each configured by their respective environment.

See Also:

Specifying Environments for MLE Modules in *JavaScript Developer's Guide* for more information about adding MLE environments for MLE modules

20.5.8.3 Managing JavaScript MLE Modules

You can use SQL DDL statements to create MLE modules as schema objects, provided you have the right privileges. You can use DDL statements to add, alter, or drop JavaScript modules.

Use a `CREATE MLE MODULE` DDL statement to create an MLE module:

Use the `ALTER MLE MODULE` DDL statement to alter attributes of existing modules.

Use the `DROP MLE MODULE` DDL statement to drop modules.

Example 20-1 Creating an MLE Module

```
CREATE MLE MODULE scott."jsmodule"
 LANGUAGE JAVASCRIPT
 AS export function func() { ... }
```

Example 20-2 Replacing an MLE Module

```
CREATE OR REPLACE MLE MODULE scott."jsmodule"
 LANGUAGE JAVASCRIPT
 AS export function func() { ... }
```

Example 20-3 Dropping an MLE Module

```
DROP MLE MODULE scott."jsmodule"
```

See Also:

- Using JavaScript Modules in MLE in *JavaScript Developer's Guide* for more information about managing JavaScript MLE modules
- System and Object Privileges Required for Working with JavaScript in MLE in *JavaScript Developer's Guide* for more information about MLE privileges

20.5.8.3.1 Built-in JavaScript MLE Modules

MLE provides a set of built-in modules that reside in the `SYS` schema. Examples include `mle-js-oracledb` (MLE JavaScript SQL driver), `mle-js-bindings` (used to exchange values between PL/SQL and dynamic MLE contexts), and `mle-js-plsqltypes` (SQL wrapper types definition).

The built-in MLE modules are available for import in any execution context. These modules are not deployed to the database but included as part of the MLE runtime.

See Also:

Built-in MLE modules API Documentation

20.5.8.4 Running JavaScript Code Using MLE Modules

You can use JavaScript code in an MLE module in the following ways:

- You can create call specifications to publish the functions that are exported from an MLE module. You can call these MLE functions like you call PL/SQL functions and procedures from SQL and PL/SQL.
- You can use the import statement to import a JavaScript MLE module into other MLE code that is written in JavaScript.

See Also:

- Calling MLE JavaScript Functions
- Importing JavaScript MLE Modules

20.5.8.4.1 Calling MLE JavaScript Functions

From SQL and PL/SQL, you can use MLE call specifications to publish the MLE JavaScript functions. You can call an MLE function from anywhere that you can call a regular PL/SQL function or procedure.

Here is an example:

```
create or replace mle module example_module language javascript as
export function string2obj(inputString) {
    if ( inputString === undefined ) {
        throw `must provide a string in the form of key1=value1;...;keyN=valueN`;
    }

    let myObject = {};

    if ( inputString.length === 0 ) {
        return myObject;
    }

    const kvPairs = inputString.split(";");

    kvPairs.forEach( pair => {
        const tuple = pair.split("=");
        if ( tuple.length === 1 ) {
            tuple[1] = false;
        } else if ( tuple.length != 2 ) {
            throw "parse error: you need to use exactly one '=' between key and value
and not use '=' in either key or value";
        }
        myObject[tuple[0]] = tuple[1];
    });
    return myObject;
}
```

```
/


create or replace function p_string_to_JSON(p_str varchar2) return JSON
as mle module example_module signature 'string2obj(string)';
/


declare
    l_json   JSON;
    l_string VARCHAR2(100);

begin
    l_string := 'a=1;b=2;c=3;d';
    l_json := p_string_to_JSON(l_string);
    dbms_output.put_line(json_serialize(l_json PRETTY));
end;
/
```

Here is the output:

```
{
  "a" : "1",
  "b" : "2",
  "c" : "3",
  "d" : false
}
```

See Also:

Calling MLE JavaScript Functions in *Oracle Database JavaScript Developer's Guide*

20.5.8.4.2 Importing JavaScript MLE Modules

You can reuse the functionality in an MLE module in other MLE language code that is outside the MLE module. MLE language code in an existing execution context can import the code of an MLE module (if the code is in the same MLE language) using the language's native import mechanism for example, `import()` in JavaScript.

Module imports in MLE include:

- Importing module functionality in an MLE module into an execution context of another MLE module
- Importing an MLE module into a code snippet to be run in a dynamic MLE execution context

Here is an example of an MLE module importing an MLE function from another MLE module:

```
create or replace mle env default_export_env
    imports ('defaultExportModule' module default_export_module);

create mle module default_import_module language javascript as
import myMathClass from "defaultExportModule";

export function mySum(){
    const result = myMathClass.sum(4, 2);
    console.log(`the sum of 4 and 2 is ${result}`);
}
/
```

See Also:

Overview of Importing MLE JavaScript Modules in *JavaScript Developer's Guide*

20.5.9 Invoking JavaScript Using Dynamic MLE Execution

This section provides a brief background of dynamic MLE execution and how you can use it to invoke JavaScript in the database.

For the complete documentation related to dynamic MLE execution, see *Oracle Database JavaScript Developer's Guide*.

Topics:

- Dynamic MLE Execution Overview
- Using Dynamic MLE Execution contexts
- Specifying an Environment for Dynamic MLE Contexts
- Running JavaScript Code Using Dynamic MLE Execution

20.5.9.1 Dynamic MLE Execution Overview

Note:

Dynamic MLE execution is suitable for developers working on frameworks (APEX) and server technology (REPL). For all other use cases, using MLE modules and environments to run JavaScript is highly recommended.

As an alternative to using JavaScript MLE modules, the dynamic MLE execution method enables you to run a JavaScript code snippet directly using the `DBMS_MLE` package without having to deploy it as an MLE module first. Dynamic MLE execution is analogous to the `DBMS_SQL` package that is used to execute dynamic SQL.

With dynamic MLE execution, you can invoke a JavaScript code snippet without storing the JavaScript code in the database. The code is not deployed as an MLE module, but is instead provided as `VARACHAR2` or `CLOB` (for larger amounts of code). The code is passed to the `DBMS_MLE` package, which then evaluates and runs the code.

Values can be passed between PL/SQL and dynamic MLE snippets by reading and writing global variables in the appropriate execution context using functions provided in the `DBMS_MLE` package.

See Also:

- Running JavaScript Code Using Dynamic MLE Execution
- Overview of Dynamic MLE Execution in *JavaScript Developer's Guide*

20.5.9.2 Using Dynamic MLE Execution contexts

MLE enables you to have explicit control over which execution context to use for each dynamic MLE snippet. Dynamic MLE execution contexts are created explicitly using the `DBMS_MLE.create_context()` function.

```
FUNCTION create_context(
  environment IN VARCHAR2 DEFAULT NULL
) RETURN context_handle_t;
```

Here are some important points to note while using dynamic MLE execution contexts:

- You can have multiple dynamic MLE snippets evaluated in the same execution context.
- Snippets evaluated in the same context have access to all global variables in the context.
- Snippets evaluated in one context have their execution state completely isolated from snippets evaluated in another context.
- Each dynamic MLE context is created on behalf of a specific database user.
- A dynamic MLE context handle identifies an execution context. This execution context is exclusively used for actions performed on this handle.
- You can create a dynamic execution context using the `DBMS_MLE.create_context()` function, which returns an execution context on behalf of the calling user.
- All MLE code evaluated in a dynamic MLE context executes with the privileges of the user on whose behalf the context was created.
- Each context is bound to a single session. Dynamic MLE snippets in different sessions cannot share runtime state.

See Also:

About MLE Execution Contexts in *JavaScript Developer's Guide* for more information about dynamic MLE execution contexts

20.5.9.3 Specifying an Environment for Dynamic MLE Contexts

When creating a dynamic MLE context, you can mention the schema name of an environment object as an optional parameter:

```
DECLARE
  ctx DBMS_MLE.context_handle_t;
BEGIN
  ctx := DBMS_MLE.create_context(environment => 'SCOTT.myenv');
  ...
END;
```

If the environment parameter to the `DBMS_MLE.create_context` function is omitted, the code running in the execution context can import only the MLE built-in modules.

Note:

The environment name passed as a string to the `DBMS_MLE.create_context` function is case sensitive and must be a valid schema name. The environment name is not implicitly converted to uppercase.

See Also:

JavaScript Developer's Guide for more information about environments.

20.5.9.4 Running JavaScript Code Using Dynamic MLE Execution

You can run JavaScript code snippets using the procedures that are defined in the `DBMS_MLE` PL/SQL package.

Using the `DBMS_MLE` package, you can:

- Create dynamic MLE execution contexts.
- Provide JavaScript code snippets as character strings in `VARCHAR2` or `CLOB`.
- Run the code in a dynamic MLE execution context.
- Pass the variables between the PL/SQL and MLE engines; export PL/SQL values to a dynamic MLE execution context, which are then available to code snippets running in that context and import values from a dynamic MLE execution context into PL/SQL.
- Deallocate the execution context.

Dynamic MLE snippets can import MLE modules using module-to-module import and execute SQL statements. You must have the `EXECUTE DYNAMIC MLE` privilege to call procedures and functions in the `DBMS_MLE` package. Dynamic MLE execution contexts created using `DBMS_MLE` are created with the privileges of the calling user. Users with the `EXECUTE` privilege on `DBMS_SYS_MLE` also have access to the `DBMS_SYS_MLE` package, which allows these users to create dynamic MLE execution contexts on behalf of any database user.

You can call the `DBMS_MLE.create_context()` function to create an execution context for dynamic MLE code. The function takes an optional parameter that specifies language options, and returns a handle to the created context.

See Also:

- Overview of Dynamic MLE Execution in *JavaScript Developer's Guide* for more information about dynamic MLE execution
- Privileges for Working with JavaScript in MLE

20.5.9.4.1 Running JavaScript Code Inline

The following example creates a dynamic MLE execution context and runs a JavaScript snippet in the context using the Q-Quote operator:

```
DECLARE
    l_ctx     dbms_mle.context_handle_t;
    l_snippet CLOB;
BEGIN
    l_ctx := dbms_mle.create_context();
    l_snippet := q'~
// the q-quote operator allows for much more readable code
console.log(`The use of the q-quote operator`);
console.log(`greatly simplifies provision of code inline`);
~';
    dbms_mle.eval(
                 l_ctx,
                 'JAVASCRIPT',
                 l_snippet
    );
    dbms_mle.drop_context(l_ctx);
EXCEPTION
    WHEN OTHERS THEN
        dbms_mle.drop_context(l_ctx);
        RAISE;
END;
/
```

20.5.9.4.2 Running JavaScript Code Using Files

Another method for providing JavaScript code is to read a `CLOB` using a `BFILE` operator, with PL/SQL code segregated from JavaScript code.

See Also:

Loading JavaScript Code from Files in *JavaScript Developer's Guide* for examples showing how to use files to run JavaScript.

20.5.10 Privileges for Working with JavaScript in MLE

MLE protects access to MLE features using database privileges, which it adds or reuses, as appropriate. Administrators can grant the necessary privileges to users, roles, or both. The minimum privilege that is required to work with MLE JavaScript code is the right to execute JavaScript code in the database. Privileges that allow users to create, alter, or drop MLE schema objects must be restrictive, as must be those with the `ANY` keyword.

To enable you to run any JavaScript code in your own schema, the following object grant must have been issued to your user account:

```
GRANT EXECUTE ON JAVASCRIPT TO <role | user>
```

To include dynamic execution of JavaScript using `DBMS_MLE`, an additional privilege is required:

```
GRANT EXECUTE DYNAMIC MLE TO <role | user>
```

Note:

When granting privileges, MLE distinguishes between dynamic MLE execution based on the `DBMS_MLE` package and the MLE execution based on MLE modules and environments.

See Also:

- MLE User Privileges for commonly used MLE privileges
- System and Object Privileges Required for Working with JavaScript in MLE in *JavaScript Developer's Guide* for more information about granting MLE privileges
- MLE Security in *JavaScript Developer's Guide* for information about other security options available for working with MLE

20.5.10.1 MLE User Privileges

Privilege Name	Description
`CREATE MLE`	Create or replace modules or environments in your schema
`CREATE ANY MLE`	Create or replace modules or environments in any schema
`ALTER ANY MLE`	Alter modules or environments in any schema
`DROP ANY MLE`	Drop modules or environments in any schema
`CREATE PROCEDURE`	Create or replace call specification in your schema
`CREATE ANY PROCEDURE`	Create or replace call specification in any schema
`DROP ANY PROCEDURE`	Drop call specification in any schema
`EXECUTE DYNAMIC MLE`	Execute any Dynamic MLE functionality
`EXECUTE ON <mle-call-specification>`	Execute the named MLE call specification
`EXECUTE ON <mle-language>`	Execute call specifications or dynamic MLE snippets in the named MLE language
`EXECUTE ON <mle-module>`	Execute call specifications against the referenced MLE module, or define the module as import in an environment
`EXECUTE ON <mle-env>`	Use the named MLE environment to configure a module context or a dynamic MLE context

20.5.11 Other Supported MLE Features

MLE (JavaScript) supports the following other features:

Calling SQL and PL/SQL from the MLE JavaScript SQL Driver

An MLE JavaScript SQL driver enables you to use JavaScript to execute SQL statements and PL/SQL blocks from within JavaScript code. JavaScript has a JavaScript-specific SQL driver API. This API is closely aligned with the API of the corresponding client-side driver. For example, the JavaScript MLE SQL Driver provides an API that is similar to the API of the node-oracledb add-on for Node.js.

See Also:

Calling PL/SQL and SQL from MLE SQL Driver in *JavaScript Developer's Guide*

Access to `stdout` and `stderr` from JavaScript

MLE provides the functionality to access data written to standard output and error streams from JavaScript code. Within a database session, these streams can be controlled individually for each database user, MLE module, and dynamic MLE context. Information that the executed JavaScript code writes to `stdout` and `stderr` is often valuable for debugging and analysis. The `DBMS_MLE` package allows for mapping `stdout` and `stderr` either to `DBMS_OUTPUT`, or to a user-provided `CLOB`.

See Also:

Access to stdout and stderr from JavaScript in *JavaScript Developer's Guide*

SODA Collections in MLE JavaScript Code

You can use any Simple Oracle Document Access (SODA) implementation to perform create, read, update, and delete (CRUD) operations on documents of nearly any kind (including video, image, sound, and other binary content). SODA is a set of NoSQL-style APIs that let you create and store collections of documents (in particular JSON) in Oracle Database, and also let you retrieve them, and query them.

See Also:

Working with SODA Collections in MLE JavaScript Code in *JavaScript Developer's Guide*

Post-execution Debugging of MLE JavaScript Modules

MLE provides the option to perform post-execution debugging on your JavaScript source code in addition to the standard print debugging. Post-execution debugging allows efficient collection of runtime state during program execution. Once execution of the code has completed, the collected data can be used to analyze program behavior and discover bugs that require attention.

See Also:

Post-Execution Debugging of MLE JavaScript Modules in *JavaScript Developer's Guide*

20.6 Choosing PL/SQL, Java, or JavaScript

Oracle Database supports PL/SQL, Java, and JavaScript on the server-side and enables you to embed SQL and PL/SQL blocks in your applications using precompilers and APIs on the client-side. To derive maximum benefits from Oracle Database, programmers must choose server-side programming with Oracle Database. With the application logic executing closer to the data, you can avoid network round trips, reuse code with centrally stored functions, and impose business rules within the database to ensure conformity in every application.

Choosing the language to use for your application depends mainly on the type of your application or project, and how you can use a language to your advantage. PL/SQL, Java, and JavaScript have their individual strengths, and they complement each other when used with Oracle Database.

PL/SQL, Java, and JavaScript inter-operate seamlessly in the server through SQL and PL/SQL call signatures. When developing with PL/SQL, you can invoke Java or JavaScript code or reference Java or JavaScript functions from PL/SQL. When developing with Java, you can invoke PL/SQL packages from distributed CORBA or Enterprise Java Beans clients. When developing with JavaScript, you can call PL/SQL and SQL from within JavaScript using the Multi-lingual Engine (MLE) JavaScript SQL Driver (`mle-js-oracledb`).

PL/SQL packages have their Java and JavaScript equivalents as shown in Table 20-1.

Table 20-1 PL/SQL Packages and Their Java and JavaScript Equivalents

PL/SQL Package	Java Equivalent	JavaScript Equivalent
`DBMS_ALERT`	Call package with JDBC.	Call with `mle-js-oracledb`.
`DBMS_DDL`	JDBC has this functionality.	`mle-js-oracledb` has this functionality.
DBMS_JOB	Schedule a job that has a Java stored subprogram.	Scheduling a job using JavaScript requires a call to `DBMS_SCHEDULER.create_job()` using the built-in JavaScript SQL driver.
`DBMS_LOCK`	Call with JDBC.	Call with `mle-js-oracledb`.
`DBMS_MAIL`	Use JavaMail.	Call with `mle-js-oracledb`.
`DBMS_OUTPUT`	Use subclass `oracle.aurora.rdbms.OracleDBMSOutputStream` or Java stored subprogram `DBMS_JAVA.SET_STREAMS`.	Use `console.log`.
`DBMS_PIPE`	Call with JDBC.	Call with `mle-js-oracledb`.

Table 20-1 (Cont.) PL/SQL Packages and Their Java and JavaScript Equivalents

PL/SQL Package	Java Equivalent	JavaScript Equivalent
`DBMS_SESSION`	Use JDBC to run an `ALTER SESSION` statement.	In most cases, you can use `mle-js-oracledb` to run an `ALTER SESSION` statement. For example: `` myConnection.execute(`alter session set events 'sql_trace wait=true'`); `` **Note:** Calling `ALTER SESSION` directly may require additional privileges to change settings, such as session attributes and set events.
`DBMS_SNAPSHOT`	Call with JDBC.	Call with `mle-js-oracledb`.
`DBMS_SQL`	Use JDBC.	Use `mle-js-oracledb`.
`DBMS_TRANSACTION`	Use JDBC to run an `ALTER SESSION` statement.	Use `mle-js-oracledb` to run an `ALTER SESSION` statement.
`DBMS_UTILITY`	Call with JDBC.	Call with `mle-js-oracledb`.

Table 20-1 (Cont.) PL/SQL Packages and Their Java and JavaScript Equivalents

PL/SQL Package	Java Equivalent	JavaScript Equivalent
`UTL_FILE`	Grant the `JAVAUSERPRIV` privilege and then use Java I/O entry points.	Call with `mle-js-oracledb`.

Note:

The `DBMS_JOB` package has been superseded by the `DBMS_SCHEDULER` package, and support for `DBMS_JOB` might be removed in future releases of Oracle Database. In particular, if you are administering jobs to manage system load, you are encouraged to disable `DBMS_JOB` by revoking the package execution privilege for users.

For more information, see DBMS_SCHEDULER and "Moving from `DBMS_JOB` to `DBMS_SCHEDULER`" in the *Oracle Database Administrator's Guide*.

Topics:

- Similarities of PL/SQL, Java, and JavaScript
- Advantages of PL/SQL
- Advantages of Java
- Advantages of JavaScript

20.6.1 Similarities of PL/SQL, Java, and JavaScript

Object-oriented features are common in PL/SQL, Java, and JavaScript as explained here:

- Java is class-oriented and JavaScript has class-based inheritance. PL/SQL does not have an explicit class concept, yet classes can be defined as object types that provide all the features of object-orientation, including constructors, inheritance, polymorphism, and substitution.
- PL/SQL has **type evolution**, which is the ability to change methods and attributes of a type while preserving subtypes and table data that use the type.
- Java has polymorphism and component models for developing distributed systems.

Other common aspect of these languages is the packages and libraries that each of them provides to enable you to execute your programs effectively and efficiently.

20.6.2 Advantages of PL/SQL

If you want to build very secure, easy-to-maintain, high performing applications on top of Oracle Database, you must use PL/SQL. As an extension of SQL, PL/SQL supports all SQL data types, data encapsulation, information hiding, overloading, and exception-handling.

The following are the advantages of using PL/SQL:

- SQL data types are easy to use in PL/SQL, with no data type conversions required.

- SQL operations are fast with PL/SQL, especially when a large amount of data is involved, or data validation requirements of your application are high.
- Code development is usually fast in PL/SQL (subject to the development tool or development environment in use). PL/SQL provides package-based stored procedures and functions that perform SQL operations on tables, thereby reducing the network traffic between a program and the database, and saving time for data intensive applications.
- Centralizing application logic provides enhanced security and productivity through built-in APIs, thereby simplifying complex data structures and helping implement security features.
- You can use Pl/SQL to build robust applications using the low-code development platform: Oracle APEX.
- There is a large user base with Oracle-supplied packages and third party libraries that you can draw upon for development.

Note:

Some advanced PL/SQL capabilities are unavailable for Java in Oracle9*i* (for example, autonomous transactions and the dblink facility for remote databases).

20.6.3 Advantages of Java

You can use Java if:

- Your application must interact with ERP systems, RMI servers, Java/J2EE, or web services.
- You must develop part of your application in the middle-tier for any the following reasons:
  - Your business logic is complex or compute-intensive with little to moderate direct SQL access.
  - You plan to implement a middle-tier-driven presentation logic.
  - Your application requires transparent Java persistence.
  - Your application requires container-managed infrastructure services.

Thus, when you need to partition your application between the database tier and middle tier, migrate that part of your application, as needed, to the middle tier and use Java/J2EE.

The following are the advantages of using Java:

- You can use Java to create component-based, network-centric application that you can easily update as business needs change.
- You can use Java for open distributed applications, and benefit from many Java-based development tools that are available throughout the industry.
- Java has native mechanisms. For example, Java has built-in security mechanisms, an automatic Garbage Collector, type safety mechanisms, a byte-code verifier, and Java 2 security.
- Java provides built-in rapid development features, such as built-in automatic bounds checking on arrays, built-in network access classes, and APIs that contain many useful and ready-to-use classes.
- Java has a vast set of class libraries, tools, and third-party class libraries that can be reused in the database.

- Java can use CORBA (which can have many different computer languages in its clients) and Enterprise Java Beans. You can invoke PL/SQL packages from CORBA or Enterprise Java Beans clients.
- You can run XML tools, the Internet File System, or JavaMail from Java.
- You can use Oracle Java Virtual Machine (JVM) for in-place data processing (calling out web services, Hadoop servers, third-party databases, and legacy systems), for running third-party Java libraries, or for running Java-based languages (Jython, Groovy Kotlin, Clojure, Scala, JRuby).
- Java is a robust language in terms of security and can therefore be safely used within the database.

20.6.4 Advantages of JavaScript

If you are looking at ease of scripting to develop end-to-end application with fast execution, JavaScript is your preferred language. With MLE support, you can run stored procedures and user-defined functions written in JavaScript within Oracle Database.

The following are the advantages of using JavaScript:

- With MLE, server-side JavaScript execution is tightly integrated with the database, with code executing within the process of a database session, close to the data and SQL execution. This integration enables efficient data exchange between JavaScript and SQL and PL/SQL.
- When you use JavaScript with Oracle Database, you benefit from the Just-In-Time (JIT) compiler offered by GraalVM, which provides self-optimizing code; so the code adapts to the data that is flowing through it.
- You can use your existing JavaScript code and run them directly against Oracle database without writing the logic in PL/SQL.
- You can create a stored procedure that references a JavaScript module for a function within that module. Alternatively, you can create a stored procedure with the JavaScript code itself inlined within the `CREATE PROCEDURE` or `CREATE FUNCTION` DDL statement.
- JavaScript has the support of a huge community that maintains a vast ecosystem of open-source packages, which you can potentially integrate into your projects.
- It is easy to get started and develop with JavaScript.
- You can use JavaScript as a server-side programming language for your Oracle APEX applications.

20.7 Overview of Precompilers

Client/server programs are typically written using **precompilers**, which are programming tools that let you embed SQL statements in high-level programs written in languages such as C, C++, or COBOL. Because the client application hosts the SQL statements, it is called a **host program**, and the language in which it is written is called the **host language**.

A precompiler accepts the host program as input, translates the embedded SQL statements into standard database runtime library calls, and generates a source program that you can compile, link, and run in the usual way.

Topics:

- Overview of the Pro*C/C++ Precompiler

- Overview of the Pro*COBOL Precompiler

> **See Also:**
>
> *Oracle Database Concepts* for additional general information about Oracle precompilers

20.7.1 Overview of the Pro*C/C++ Precompiler

For the Pro*C/C++ precompiler, the host language is either C or C++. Some features of the Pro*C/C++ precompiler are:

- You can write multithreaded programs if your platform supports a threads package. Concurrent connections are supported in either single-threaded or multithreaded applications.
- You can improve performance by embedding PL/SQL blocks. These blocks can invoke subprograms in Java or PL/SQL that are written by you or provided in Oracle Database packages.
- Using precompiler options, you can check the syntax and semantics of your SQL or PL/SQL statements during precompilation, and at runtime.
- You can invoke stored PL/SQL and Java subprograms. Modules written in COBOL or in C can be invoked from Pro*C/C++. External C subprograms in shared libraries can be invoked by your program.
- You can conditionally precompile sections of your code so that they can run in different environments.
- You can use arrays, or structures, or arrays of structures as host and indicator variables in your code to improve performance.
- You can deal with errors and warnings so that data integrity is guaranteed. As a programmer, you control how errors are handled.
- Your program can convert between internal data types and C language data types.
- The Oracle Call Interface (OCI) and Oracle C++ Call Interface (OCCI), lower-level C and C++ interfaces, are available for use in your precompiler source.
- Pro*C/C++ supports dynamic SQL, a technique that enables users to input variable values and statement syntax.
- Pro*C/C++ can use special SQL statements to manipulate tables containing user-defined object types. An Object Type Translator (OTT) maps the ADTs and named collection types in your database to structures and headers that you include in your source.
- Three kinds of collection types: associative arrays, nested tables and `VARRAY`, are supported with a set of SQL statements that give you a high degree of control over data.
- Large Objects are accessed by another set of SQL statements.
- A new ANSI SQL standard for dynamic SQL is supported for new applications, so that you can run SQL statements with a varying number of host variables. An older technique for dynamic SQL is still usable by pre-existing applications.
- Globalization support lets you use multibyte characters and UCS2 Unicode data.

- Using scrollable cursors, you can move backward and forward through a result set. For example, you can fetch the last row of the result set, or jump forward or backward to an absolute or relative position within the result set.
- A **connection pool** is a group of physical connections to a database that can be shared by several named connections. Enabling the connection pool option can help optimize the performance of Pro*C/C++ application. The connection pool option is not enabled by default.

See Also:

*Pro*C/C++ Programmer's Guide* for complete information about the Pro*C/C++ precompiler

Example 20-4 is a code fragment from a C source program that queries the table `employees` in the schema `hr`.

Example 20-4   Pro*C/C++ Application

```
...
#define  UNAME_LEN   10
...
int   emp_number;
/* Define a host structure for the output values of a SELECT statement. */
/* No declare section needed if precompiler option MODE=ORACLE          */
struct {
    VARCHAR  last_name[UNAME_LEN];
    float    salary;
    float    commission_pct;
} emprec;
/* Define an indicator structure to correspond to the host output structure. */
struct {
    short emp_name_ind;
    short sal_ind;
    short comm_ind;
} emprec_ind;
...
/* Select columns last_name, salary, and commission_pct given the user's input
/* for employee_id. */
    EXEC SQL SELECT last_name, salary, commission_pct
        INTO :emprec INDICATOR :emprec_ind
        FROM employees
        WHERE employee_id = :emp_number;
...
```

The embedded `SELECT` statement differs slightly from the interactive (SQL*Plus) `SELECT` statement. Every embedded SQL statement begins with `EXEC SQL`. The colon (:) precedes every host (C) variable. The returned values of data and indicators (set when the data value is `NULL` or character columns were truncated) can be stored in structs (such as in the preceding code fragment), in arrays, or in arrays of structs. Multiple result set values are handled very simply in a manner that resembles the case shown, where there is only one result, because of the unique employee number. Use the actual names of columns and tables in embedded SQL.

Either use the default precompiler option values or enter values that give you control over the use of resources, how errors are reported, the formatting of output, and how cursors (which correspond to a particular connection or SQL statement) are managed. Cursors are used when there are multiple result set values.

Enter the options either in a configuration file, on the command line, or inline inside your source code with a special statement that begins with `EXEC ORACLE`. If there are no errors found, you can compile, link, and run the output source file, like any other C program that you write.

Use the precompiler to create server database access from clients that can be on many different platforms. Pro*C/C++ gives you the freedom to design your own user interfaces and to add database access to existing applications.

Before writing your embedded SQL statements, you can test interactive versions of the SQL in SQL*Plus and then make minor changes to start testing your embedded SQL application.

20.7.2 Overview of the Pro*COBOL Precompiler

For the Pro*COBOL precompiler, the host language is COBOL. Some features of the Pro*COBOL precompiler are:

- You can invoke stored PL/SQL or Java subprograms. You can improve performance by embedding PL/SQL blocks. These blocks can invoke PL/SQL subprograms written by you or provided in Oracle Database packages.
- Precompiler options enable you to define how cursors, errors, syntax-checking, file formats, and so on, are handled.
- Using precompiler options, you can check the syntax and semantics of your SQL or PL/SQL statements during precompilation, and at runtime.
- You can conditionally precompile sections of your code so that they can run in different environments.
- Use tables, or group items, or tables of group items as host and indicator variables in your code to improve performance.
- You can program how errors and warnings are handled, so that data integrity is guaranteed.
- Pro*COBOL supports dynamic SQL, a technique that enables users to input variable values and statement syntax.

See Also:

*Pro*COBOL Programmer's Guide* for complete information about the Pro*COBOL precompiler

Example 20-5 is a code fragment from a COBOL source program that queries the table `employees` in the schema `hr`.

Example 20-5 Pro*COBOL Application

```
...
 WORKING-STORAGE SECTION.
*
* DEFINE HOST INPUT AND OUTPUT HOST AND INDICATOR VARIABLES.
* NO DECLARE SECTION NEEDED IF MODE=ORACLE.
*
 01  EMP-REC-VARS.
     05  EMP-NAME    PIC X(10) VARYING.
     05  EMP-NUMBER  PIC S9(4) COMP VALUE ZERO.
     05  SALARY      PIC S9(5)V99 COMP-3 VALUE ZERO.
     05  COMMISSION  PIC S9(5)V99 COMP-3 VALUE ZERO.
     05  COMM-IND    PIC S9(4) COMP VALUE ZERO.
...
 PROCEDURE DIVISION.
...
     EXEC SQL
         SELECT last_name, salary, commission_pct
         INTO :EMP-NAME, :SALARY, :COMMISSION:COMM-IND
         FROM employees
         WHERE employee_id = :EMP-NUMBER
     END-EXEC.
...
```

The embedded `SELECT` statement is only slightly different from an interactive (SQL*Plus) `SELECT` statement. Every embedded SQL statement begins with EXEC SQL. The colon (:) precedes every host (COBOL) variable. The SQL statement is terminated by `END-EXEC`. The returned values of data and indicators (set when the data value is `NULL` or character columns were truncated) can be stored in group items (such as in the preceding code fragment), in tables, or in tables of group items. Multiple result set values are handled very simply in a manner that resembles the case shown, where there is only one result, given the unique employee number. Use the actual names of columns and tables in embedded SQL.

Use the default precompiler option values, or enter values that give you control over the use of resources, how errors are reported, the formatting of output, and how cursors are managed (cursors correspond to a particular connection or SQL statement).

Enter the options in a configuration file, on the command line, or inline inside your source code with a special statement that begins with EXEC ORACLE. If there are no errors found, you can compile, link, and run the output source file, like any other COBOL program that you write.

Use the precompiler to create server database access from clients that can be on many different platforms. Pro*COBOL gives you the freedom to design your own user interfaces and to add database access to existing COBOL applications.

The embedded SQL statements available conform to an ANSI standard, so that you can access data from many databases in a program, including remote servers networked through Oracle Net.

Before writing your embedded SQL statements, you can test interactive versions of the SQL in SQL*Plus and then make minor changes to start testing your embedded SQL application.

20.8 Overview of OCI and OCCI

The Oracle Call Interface (OCI) and Oracle C++ Call Interface (OCCI) are application programming interfaces (APIs) that enable you to create applications that use native subprogram invocations of a third-generation language to access Oracle Database and control all phases of SQL statement execution. These APIs provide:

- Improved performance and scalability through the efficient use of system memory and network connectivity
- Consistent interfaces for dynamic session and transaction management in a two-tier client/server or multitier environment
- *N*-tiered authentication
- Comprehensive support for application development using Oracle Database objects
- Access to external databases

- Ability to develop applications that service an increasing number of users and requests without additional hardware investments

OCI lets you manipulate data and schemas in a database using a host programming language, such as C. OCCI is an object-oriented interface suitable for use with C++. These APIs provide a library of standard database access and retrieval functions in the form of a dynamic runtime library that can be linked in an application at runtime. You need not embed SQL or PL/SQL within 3GL programs.

See Also:

For more information about OCI and OCCI calls:

- *Oracle Call Interface Programmer's Guide*
- *Oracle C++ Call Interface Programmer's Guide*
- *Oracle Database Advanced Queuing User's Guide*
- *Oracle Database Globalization Support Guide*
- *Oracle Database Data Cartridge Developer's Guide*

Topics:

- Advantages of OCI and OCCI
- OCI and OCCI Functions
- Procedural and Nonprocedural Elements of OCI and OCCI Applications
- Building an OCI or OCCI Application

20.8.1 Advantages of OCI and OCCI

OCI and OCCI provide significant advantages over other methods of accessing Oracle Database:

- More fine-grained control over all aspects of the application design.
- High degree of control over program execution.
- Use of familiar 3GL programming techniques and application development tools such as browsers and debuggers.
- Support of dynamic SQL, method 4.
- Availability on the broadest range of platforms of all the Oracle Database programmatic interfaces.
- Dynamic bind and define using callbacks.
- Describe functionality to expose layers of server metadata.
- Asynchronous event notification for registered client applications.
- Enhanced array data manipulation language (DML) capability for arrays.
- Ability to associate a commit request with a statement execution to reduce round trips.
- Optimization for queries using transparent prefetch buffers to reduce round trips.
- Thread safety, so you do not have to implement mutual exclusion (mutex) locks on OCI and OCCI handles.

- The server connection in nonblocking mode means that control returns to the OCI code when a call is still running or cannot complete.

20.8.2 OCI and OCCI Functions

Both OCI and OCCI have four kinds of functions:

Kind of Function	Purpose
Relational	To manage database access and process SQL statements
Navigational	To manipulate objects retrieved from the database
Database mapping and manipulation	To manipulate data attributes of Oracle Database types
External subprogram	To write C callbacks from PL/SQL (OCI only)

20.8.3 Procedural and Nonprocedural Elements of OCI and OCCI Applications

OCI and OCCI enable you to develop applications that combine the nonprocedural data access power of SQL with the procedural capabilities of most programming languages, including C and C++. Procedural and nonprocedural languages have these characteristics:

- In a nonprocedural language program, the set of data to be operated on is specified, but what operations are performed and how the operations are to be carried out is not specified. The nonprocedural nature of SQL makes it an easy language to learn and to use to perform database transactions. It is also the standard language used to access and manipulate data in modern relational and object-relational database systems.
- In a procedural language program, the execution of most statements depends on previous or subsequent statements and on control structures, such as loops or conditional branches, which are unavailable in SQL. The procedural nature of these languages makes them more complex than SQL, but it also makes them very flexible and powerful.

The combination of both nonprocedural and procedural language elements in an OCI or OCCI program provides easy access to Oracle Database in a structured programming environment.

OCI and OCCI support all SQL data definition, data manipulation, query, and transaction control facilities that are available through Oracle Database. For example, an OCI or OCCI program can run a query against Oracle Database. The queries can require the program to supply data to the database using input (bind) variables, as follows:

```
SELECT name FROM employees WHERE empno = :empnumber
```

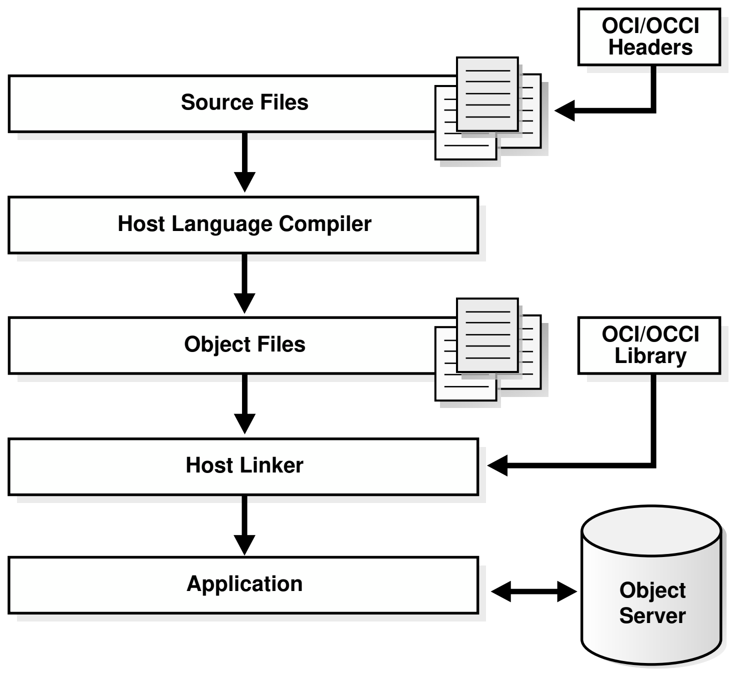
In the preceding SQL statement, `:empnumber` is a placeholder for a value to be supplied by the application.

Alternatively, you can use PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written in SQL alone. OCI and OCCI also provide facilities for accessing and manipulating objects in Oracle Database.

20.8.4 Building an OCI or OCCI Application

As Figure 20-1 shows, you compile and link an OCI or OCCI program in the same way that you compile and link a nondatabase application. There is no need for a separate preprocessing or precompilation step.

Figure 20-1 The OCI or OCCI Development Process



Note:

To properly link your OCI and OCCI programs, it might be necessary on some platforms to include other libraries, in addition to the OCI and OCCI libraries. Check your Oracle platform-specific documentation for further information about extra libraries that might be required.

20.9 Comparison of Precompilers and OCI

Precompiler applications typically contain less code than equivalent OCI applications, which can help productivity.

Some situations require detailed control of the database and are suited for OCI applications (either pure OCI or a precompiler application with embedded OCI calls):

- OCI provides more detailed control over multiplexing and migrating sessions.
- OCI provides dynamic bind and define using callbacks that can be used for any arbitrary structure, including lists.
- OCI has many calls to handle metadata.
- OCI enables asynchronous event notifications to be received by a client application. It provides a means for clients to generate notifications for propagation to other clients.
- OCI enables DML statements to use arrays to complete as many iterations as possible before returning any error messages.
- OCI calls for special purposes include Advanced Queuing, globalization support, Data Cartridges, and support of the date and time data types.

- OCI calls can be embedded in a Pro*C/C++ application.

20.10 Overview of Oracle Data Provider for .NET (ODP.NET)

Oracle Data Provider for .NET (ODP.NET) is an implementation of a data provider for Oracle Database.

ODP.NET uses APIs native to Oracle Database to offer fast and reliable access from any .NET application to database features and data. It also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library.

For programmers using Oracle Provider for OLE DB, ADO (ActiveX Data Objects) provides an automation layer that exposes an easy programming model. ADO.NET provides a similar programming model, but without the automation layer, for better performance. More importantly, the ADO.NET model enables native providers such as ODP.NET to expose specific features and data types specific to Oracle Database.

See Also:

Oracle Data Provider for .NET Developer's Guide for Microsoft Windows

This is a simple C# application that connects to Oracle Database and displays its version number before disconnecting:

```
using System;
using Oracle.DataAccess.Client;

class Example
{
  OracleConnection con;

  void Connect()
  {
    con = new OracleConnection();
    con.ConnectionString = "User Id=hr;Password=password;Data Source=oracle";
    con.Open();
    Console.WriteLine("Connected to Oracle" + con.ServerVersion);
  }

  void Close()
  {
    con.Close();
    con.Dispose();
  }

  static void Main()
  {
    Example example = new Example();
    example.Connect();
    example.Close();
  }
}
```

Note:

Additional samples are provided in directory `ORACLE_BASE\ORACLE_HOME\ODP.NET\Samples`.

20.11 Overview of OraOLEDB

Oracle Provider for OLE DB (OraOLEDB) is an OLE DB data provider that offers high performance and efficient access to Oracle data by OLE DB consumers. In general, this developer's guide assumes that you are using OraOLEDB through OLE DB or ADO.

See Also:

Oracle Provider for OLE DB Developer's Guide for Microsoft Windows