1

Introduction to Java in Oracle Database

Java in Oracle Database is also known as Oracle JVM. You can use Oracle JVM for in-place data processing; calling out Web-Services, Hadoop servers, third-party databases, and legacy systems; running third-party Java libraries; or, running Java-based languages such as Jython, Groovy Kotlin, Clojure, Scala, and JRuby.

Oracle JVM is also used by database components such as AQ JMS, XDB, Spatial, Scheduler, and Java XA. This chapter provides an overview of Oracle JVM, which starts with a basic introduction to the Java language to Oracle PL/SQL developers, who are accustomed to developing server-side applications that are integrated with SQL data. You can develop server-side Java applications that take advantage of the scalability and performance of Oracle Database.

This chapter contains the following sections:

1.1 Overview of Java

Java has emerged as the object-oriented programming language of choice. Some of the important concepts of Java include:

- A Java virtual machine (JVM), which provides the fundamental basis for platform independence
- Automated storage management techniques, such as garbage collection
- Language syntax that is similar to that of the C language

The result is a language that is object-oriented and efficient for application programming.

This section covers the following topics:

1.1.1 Java and Object-Oriented Programming Terminology

The following terms are common in Java application development in Oracle Database environment:

- Classes
- Objects
- Interfaces
- Encapsulation
- Inheritance
- Polymorphism

1.1.1.1 Classes

All object-oriented programming languages support the concept of a class. As with a table definition, a class provides a template for objects that share common characteristics. Each class can define the following:

- Fields

  Fields are variables that are present in each object or instance of a particular class, or are variables that are global and common to all instances. Instance fields are analogous to the columns of a relational table row. The class defines the fields and the type of each field.

  You can declare fields in Java as static. Fields of a class that are declared as static are global and common to all instances of that class. There is only one value at any given time for a static field within a given instantiation of a Java runtime. Fields that are not declared as static are created as distinct values within each instance of the class.

  The `public`, `private`, `protected`, and default access modifiers define the scope of the field in the application. The Java Language Specification (JLS) defines the rules of visibility of data for all fields. These rules define under what circumstances you can access the data in these fields.

  In the example illustrated in Figure 1-1, the employee identifier is defined as `private`, indicating that other objects cannot access this field directly. In the example, objects can access the `id` field by calling the `getId()` method.

- Methods

  Methods are procedures associated with a class. Like a field, a method can be declared as `static`, in which case it can be called globally. If a method is not declared as static, it means that the method is an instance method and can be called only as an operation on an object, where the object is an instance of the class.

  Similar to fields, methods can be declared as `public`, `private`, `protected`, or default access. This declaration defines the scope in which the method can be called.

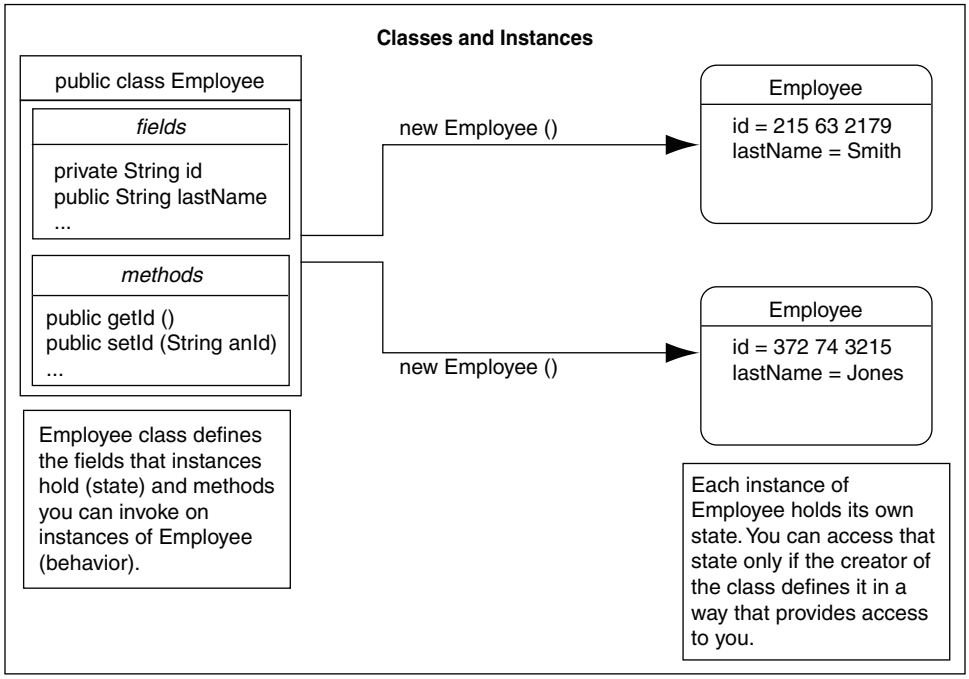
1.1.1.2 Objects

A Java object is an instance of a class and is analogous to a relational table row. Objects are collections of data values, the individual elements of which are described by the non-static field definitions of the class.

Figure 1-1 shows an example of an `Employee` class defined with two fields, `id`, which is the employee identifier, and `lastName`, which is the last name of the employee, and the `getId()`

and `setId(String anId)` methods. The `id` field is `private`, and the `lastName` field, the `getId()` method and the `setId(String anId)` method are `public`.

Figure 1-1 Classes and Instances



When you create an instance, the fields store individual and private information relevant only to the employee. That is, the information contained within an employee instance is known only to that particular employee. The example in Figure 1-1 shows two instances of the `Employee` class, one for the employee Smith and one for Jones. Each instance contains information relevant to the individual employee.

1.1.1.3 Modules

Starting from JDK 9, if a set of packages is sufficiently cohesive, then the packages may be grouped into a module.

A module categorizes some or all of its packages as exported, which means that their classes and interfaces may be accessed from code outside the module. If a package is not exported by a module, then only code inside the module may access its classes and interfaces.

> **See Also:**
>
> Java Language Specification

1.1.1.4 Inheritance

Inheritance is an important feature of object-oriented programming languages. It enables classes to include properties of other classes. The class that inherits the properties is called a child class or subclass, and the class from which the properties are inherited is called a parent class or superclass. This feature also helps in reusing already defined code.

In the example illustrated in Figure 1-1, you can create a `FullTimeEmployee` class that inherits the properties of the `Employee` class. The properties inherited depend on the access modifiers declared for each field and method of the superclass.

1.1.1.5 Interfaces

Java supports only single inheritance, that is, each class can inherit fields and methods of only one class. If you need to inherit properties from more than one source, then Java provides the concept of interfaces, which is a form of multiple inheritance. Interfaces are similar to classes. However, they define only the signature of the methods and not their implementations. The methods that are declared in the interface are implemented in the classes. Multiple inheritance occurs when a class implements multiple interfaces.

1.1.1.6 Encapsulation

Encapsulation describes the ability of an object to hide its data and methods from the rest of the world and is one of the fundamental principles of object-oriented programming. In Java, a class encapsulates the fields, which hold the state of an object, and the methods, which define the actions of the object. Encapsulation enables you to write reusable programs. It also enables you to restrict access only to those features of an object that are declared `public`. All other fields and methods are `private` and can be used for internal object processing.

In the example illustrated in Figure 1-1, the `id` field is `private`, and access to it is restricted to the object that defines it. Other objects can access this field using the `getId()` method. Using encapsulation, you can deny access to the `id` field either by declaring the `getId()` method as `private` or by not defining the `getId()` method.

1.1.1.7 Polymorphism

Polymorphism is the ability for different objects to respond differently to the same message. In object-oriented programming languages, you can define one or more methods with the same name. These methods can perform different actions and return different values.

In the example in Figure 1-1, assume that the different types of employees must be able to respond with their compensation to date. Compensation is computed differently for different types of employees:

- Full-time employees are eligible for a bonus.
- Non-exempt employees get overtime pay.

In procedural languages, you write a `switch` statement, with the different possible cases defined, as follows:

```
switch: (employee.type)
{
  case: Employee
        return employee.salaryToDate;
  case: FullTimeEmployee
        return employee.salaryToDate + employee.bonusToDate
  ...
}
```

If you add a new type of employee, then you must update the `switch` statement. In addition, if you modify the data structure, then you must modify all `switch` statements that use it. In an object-oriented language, such as Java, you can implement a method, `compensationToDate()`, for each subclass of the `Employee` class, if it contains information beyond what is already

defined in the `Employee` class. For example, you could implement the `compensationToDate()` method for a non-exempt employee, as follows:

```
public float compensationToDate()
{
  return (super.compensationToDate() + this.overtimeToDate());
}
```

For a full-time employee, the `compensationToDate()` method can be implemented as follows:

```
public float compensationToDate()
{
  return (super.compensationToDate() + this.bonusToDate());
}
```

This common use of the method name enables you to call methods of different classes and obtain the required results, without specifying the type of the employee. You do not have to write specific methods to handle full-time employees and part-time employees.

In addition, you can create a `Contractor` class that does not inherit properties from `Employee` and implements a `compensationToDate()` method in it. A program that calculates total payroll to date would iterate over all people on payroll, regardless of whether they were full-time or part-time employees or contractors, and add up the values returned from calling the `compensationToDate()` method on each. You can safely make changes to the individual `compensationToDate()` methods or the classes, and know that callers of the methods will work correctly.

1.1.2 Key Features of the Java Language

The Java language provides certain key features that make it ideal for developing server applications. These features include:

- Simplicity

  Java is simpler than most other languages that are used to create server applications, because of its consistent enforcement of the object model. The large, standard set of class libraries brings powerful tools to Java developers on all platforms.

- Portability

  Java is portable across platforms. It is possible to write platform-dependent code in Java, and it is also simple to write programs that move seamlessly across systems.

  > **See Also:**
  >
  > " Java Virtual Machine"

- Automatic storage management

  A JVM automatically performs all memory allocation and deallocation while the program is running. Java programmers cannot explicitly allocate memory for new objects or free memory for objects that are no longer referenced. Instead, they depend on a JVM to perform these operations. The process of freeing memory is known as garbage collection.

- Strong typing

  Before you use a field, you must declare the type of the field. Strong typing in Java makes it possible to provide a reasonable and safe solution to interlanguage calls between Java and PL/SQL applications, and to integrate Java and SQL calls within the same application.

- No pointers

  Although Java is quite similar to C in its syntax, it does not support direct pointers or pointer manipulation. You pass all parameters, except primitive types, by reference and not by value. As a result, the object identity is preserved. Java does not provide low level, direct access to pointers, thereby eliminating any possibility of memory corruption and leaks.

- Exception handling

  Java exceptions are objects. Java requires developers to declare which exceptions can be thrown by methods in any particular class.

- Flexible namespace

  Java defines classes and places them within a hierarchical structure that mirrors the domain namespace of the Internet. You can distribute Java applications and avoid name collisions. Java extensions, such as the Java Naming and Directory Interface (JNDI), provide a framework for multiple name services to be federated. The namespace approach of Java is flexible enough for Oracle to incorporate the concept of a schema for resolving class names in full compliance with the JLS.

- Security

  The design of Java byte codes and JVM specification allow for built-in mechanisms to verify the security of Java binary code. Oracle Database is installed with an instance of Security Manager that, when combined with Oracle Database security, determines who can call any Java methods.

- Standards for connectivity to relational databases

  Java Database Connectivity (JDBC) enables Java code to access and manipulate data in relational databases. Oracle provides drivers that allow vendor-independent, portable Java code to access the relational database.

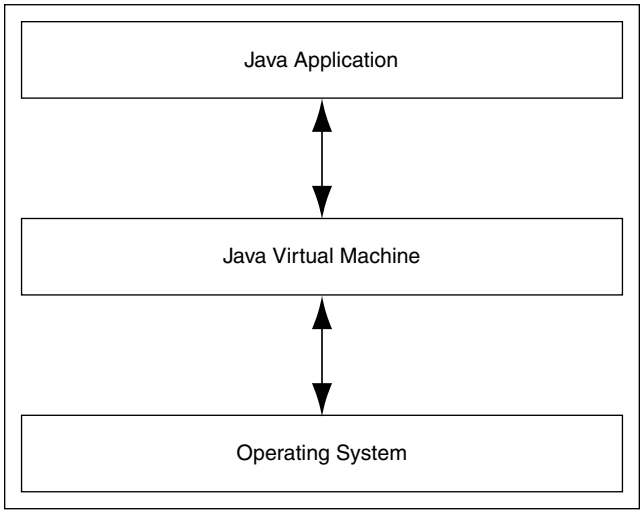
1.1.3 Java Virtual Machine

As with other high-level computer languages, the Java source compiles to low-level machine instructions. In Java, these instructions are known as bytecodes, because each instruction has a uniform size of one byte. Most other languages, such as C, compile to machine-specific instructions, such as instructions specific to an Intel or HP processor.

When compiled, the Java code gets converted to a standard, platform-independent set of bytecodes, which are executed by a Java Virtual Machine (JVM). A JVM is a separate program that is optimized for the specific platform on which you run your Java code.

Figure 1-2 illustrates how Java can maintain platform independence. Each platform has a JVM installed that is specific to the operating system. The Java bytecodes get interpreted through the JVM into the appropriate platform dependent actions.

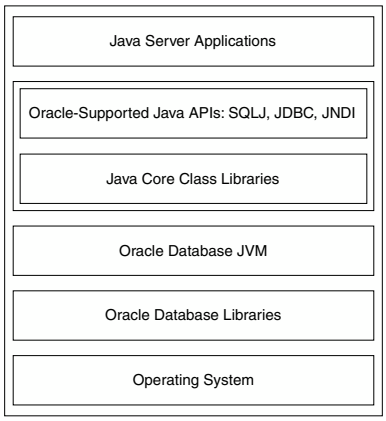
Figure 1-2 Java Component Structure



When you develop a Java application, you use predefined core class libraries written in the Java language. The Java core class libraries are logically divided into packages that provide commonly used functionality. Basic language support is provided by the `java.lang` package, I/O support is provided by the `java.io` package, and network access is provided by the `java.net` package. Together, a JVM and the core class libraries provide a platform on which Java programmers can develop applications, which will run successfully on any operating system that supports Java. This concept is what drives the "write once, run anywhere" idea of Java.

Figure 1-3 illustrates how Oracle Java applications reside on top of the Java core class libraries, which reside on top of the JVM. Because the Oracle Java support system is located within the database, the JVM interacts with Oracle Database libraries, instead of directly interacting with the operating system.

Figure 1-3 Oracle Database Java Component Structure



To know more about Java and JVM, you can refer to the Java Language Specification (JLS) and the JVM specification. The JLS defines the syntax and semantics, and the JVM specification defines the necessary low-level actions for the system that runs the application. In addition, there is also a compatibility test suite for JVM implementors to determine if they have complied with the specifications. This test suite is known as the Java Compatibility Kit (JCK).

Oracle JVM implementation complies fully with JCK. Part of the overall Java strategy is that an openly specified standard, together with a simple way to verify compliance with that standard, allows vendors to offer uniform support for Java across all platforms.

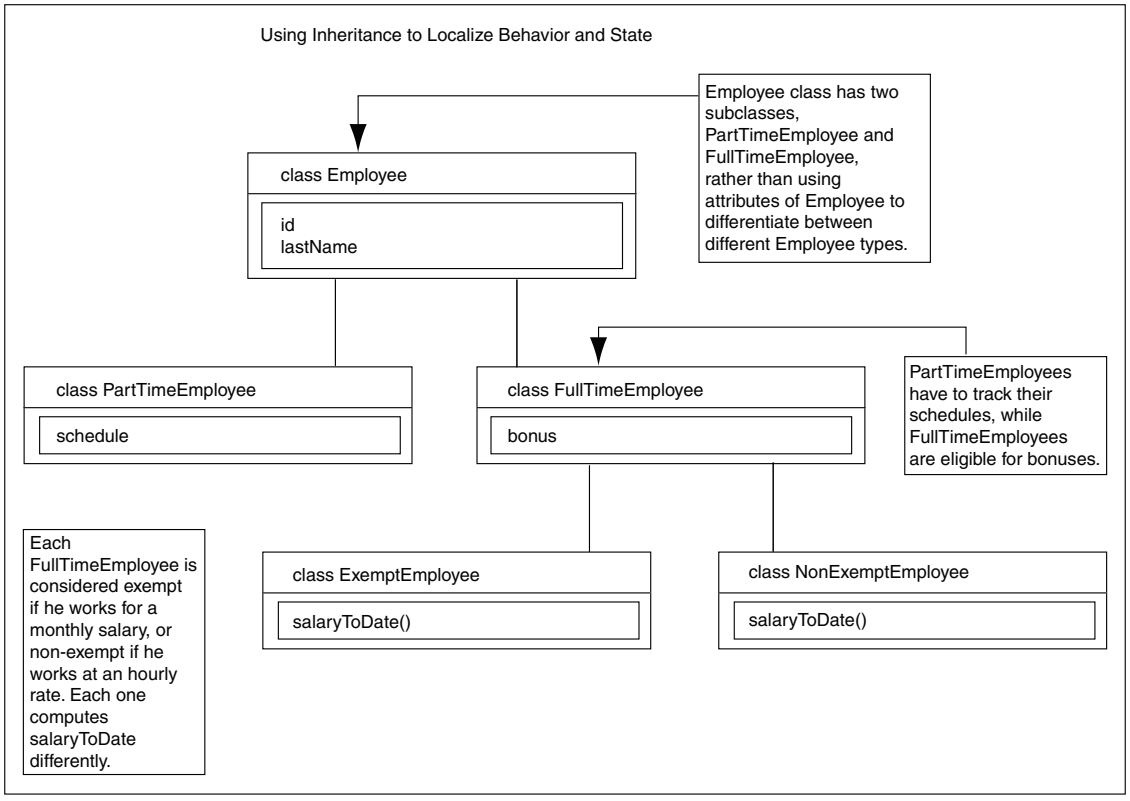
1.1.4 Java Class Hierarchy

Java defines classes within a large hierarchy of classes. At the top of the hierarchy is the `Object` class. All classes in Java inherit from the `Object` class at some level, as you walk up through the inheritance chain of superclasses. When we say Class B inherits from Class A, each instance of Class B contains all the fields defined in class B, as well as all the fields defined in Class A.

Figure 1-4 illustrates a generic Java class hierarchy. The `FullTimeEmployee` class contains the `id` and `lastName` fields defined in the `Employee` class, because it inherits from the `Employee` class. In addition, the `FullTimeEmployee` class adds another field, `bonus`, which is contained only within `FullTimeEmployee`.

You can call any method on an instance of Class B that was defined in either Class A or Class B. In the example, the `FullTimeEmployee` instance can call methods defined only in the `FullTimeEmployee` class and methods defined in the `Employee` class.

Figure 1-4 Class Hierarchy



Instances of Class B can be substituted for instances of Class A, which makes inheritance another powerful construct of object-oriented languages for improving code reuse. You can create classes that define behavior and state where it makes sense in the hierarchy, yet make use of preexisting functionality in class libraries.

1.2 About Using Java in Oracle Database

You can write and load Java applications within the database because it is a safe language with a lot of security features. Java has been developed to prevent anyone from tampering with the operating system where the Java code resides in. Some languages, such as C, can introduce security problems within the database. However, Java, because of its design, is a robust language that can be used within the database.

Although the Java language presents many advantages to developers, providing an implementation of a JVM that supports Java server applications in a scalable manner is a challenge. This section discusses the following challenges:

- Java and RDBMS: A Robust Combination
- About Multithreading
- Memory Spaces Management
- Footprint
- Performance of an Oracle JVM
- Dynamic Class Loading

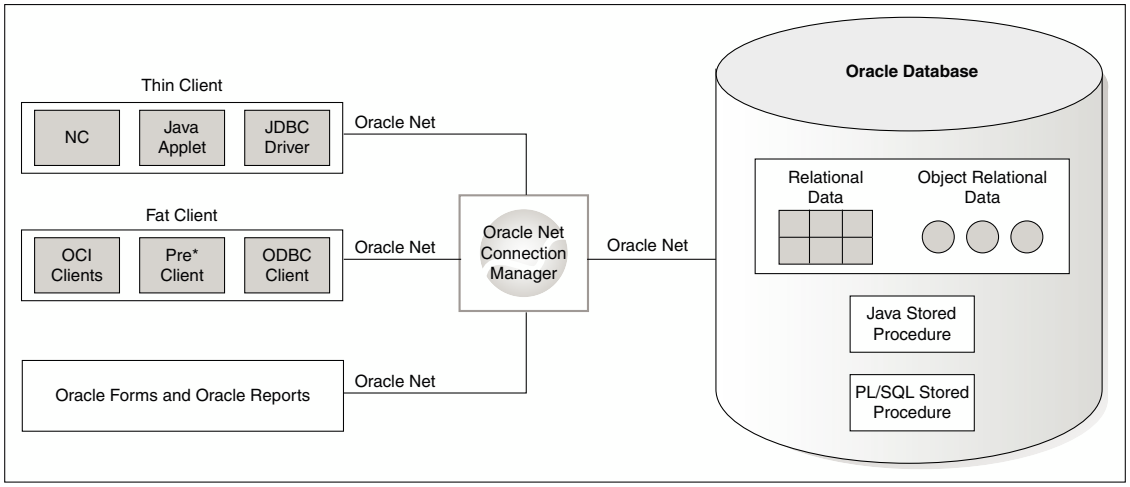
1.2.1 Java and RDBMS: A Robust Combination

Oracle Database provides Java applications with a dynamic data-processing engine that supports complex queries and different views of the same data. All client requests are assembled as data queries for immediate processing, and query results are generated dynamically.

The combination of Java and Oracle Database helps you to create component-based, network-centric applications that can be easily updated as business needs change. In addition, you can move applications and data stores off the desktop and onto intelligent networks and network-centric servers. More important, you can access those applications and data stores from any client device.

Figure 1-5 shows a traditional two-tier, client/server configuration in which clients call Java stored procedures the same way they call PL/SQL stored procedures. The figure also shows how Oracle Net Services Connection Manager can combine many network connections into a single database connection. This enables Oracle Database to support a large number of concurrent users.

Figure 1-5 Two-Tier Client/Server Configuration



1.2.2 About Multithreading

Multithreading is one of the key scalability features of the Java language. The Java language and class libraries make it simpler to write multithreaded applications in Java than many other languages, but it is still a daunting task in any language to write reliable, scalable multithreaded code.

Oracle Database server efficiently schedules work for thousands of users. The Oracle JVM takes advantage of the session architecture of Oracle database to concurrently run Java applications for hundreds to thousands of users. Although Oracle Database supports Java language-level threads required by the JLS and JCK, scalability will not increase by using threads within the scope of the database. By using the embedded scalability of the database, the need for writing multithreaded Java servers is eliminated.

You should use the facilities of Oracle Database for scheduling users by writing single-threaded Java applications. The database can schedule processes between each application, and thus, you achieve scalability without having to manage threads. You can still write multithreaded Java applications, but multiple Java threads will not increase the performance of the server.

One complication multithreading creates is the interaction of threads and automated storage management or garbage collection. The garbage collector running in a generic JVM has no knowledge of which Java language threads are running or how the underlying operating system schedules them. The difference between a non-Oracle Database model and an Oracle JVM model is as follows:

- Non-Oracle Database model

  A single user maps to a single Java thread and a single garbage collector manages all garbage from all users. Different techniques typically deal with allocation and collection of objects of varying lifetimes and sizes. The result in a heavily multithreaded application is, at best, dependent upon operating system support for native threads, which can be unreliable and limited in scalability. High levels of scalability for such implementations have not been convincingly demonstrated.

- Oracle JVM model

  Even when thousands of users connect to the server and run the same Java code, each user experiences it as if they are running their own Java code on their own JVM. The

responsibility of an Oracle JVM is to make use of operating system processes and threads and the scalable approach of Oracle Database. As a result of this approach, the garbage collector of the Oracle JVM is more reliable and efficient because it never collects garbage from more than one user at any time.

> **See Also:**
>
> "Overview of Threading in Oracle Database"

1.2.3 Memory Spaces Management

Garbage collection is a major function of the automated storage management feature of Java, eliminating the need for Java developers to allocate and free memory explicitly. Consequently, this eliminates a large source of memory leaks that are commonly found in C and C++ programs. However, garbage collection contributes to the overhead of program execution speed and footprint.

Garbage collection imposes a challenge to the JVM developer seeking to supply a highly scalable and fast Java platform. An Oracle JVM meets these challenges in the following ways:

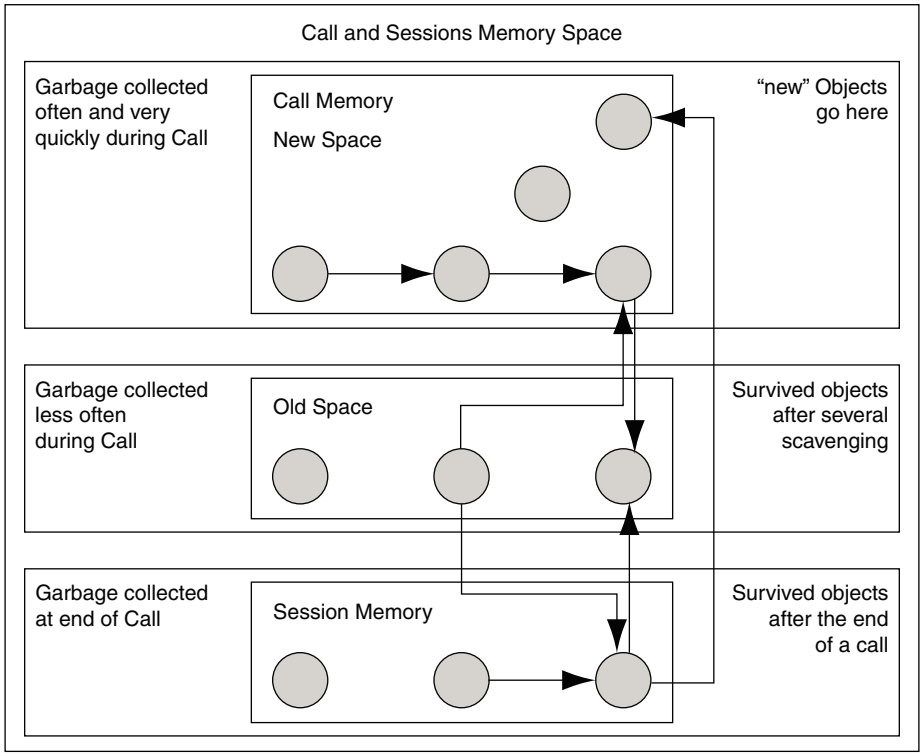
- The Oracle JVM uses Oracle Database scheduling facilities, which can manage multiple users efficiently.
- Garbage collection is performed consistently for multiple users, because garbage collection is focused on a single user within a single session. The Oracle JVM has an advantage, because the burden and complexity of the job of the memory manager does not increase as the number of users increases. The memory manager performs the allocation and collection of objects within a single session, which typically translates to the activity of a single user.
- The Oracle JVM uses different garbage collection techniques depending on the type of memory used. These techniques provide high efficiency and low overhead.

The two types of memory space are call space and session space.

Memory space	Description
Call space	It is a fast and inexpensive type of memory. It primarily exists for the length of a call. Call memory space is divided into new and old segments. All new objects are created within new memory. Objects that have survived several scavenges are moved into old memory.
Session space	It is an expensive, performance-wise memory. It primarily exists for the length of a session. All `static` fields and any objects that exist beyond the lifetime of a call exist here.

Figure 1-6 illustrates the different actions performed by the garbage collector.

Figure 1-6 Garbage Collection



Garbage collection algorithms within an Oracle JVM adhere to the following rules:

1. New objects are created within a new call space.
2. Scavenging occurs at a set interval. Some programmers create objects frequently for only a short duration. These types of objects are created and garbage-collected quickly within the new call space. This is known as **scavenging**.
3. Any objects that have survived several iterations of scavenging are considered to be objects that can exist for a while. These objects are moved out of new call space into old call space. During the move, they are also compacted. Old call space is scavenged or garbage collected less often and, therefore, provides better performance.
4. At the end of the call, any objects that are to exist beyond the call are moved into session space.

Figure 1-6 illustrates the steps listed in the preceding text. This approach applies sophisticated allocation and collection schemes tuned to the types and lifetimes of objects. For example, new objects are allocated in fast and inexpensive call memory, designed for quick allocation and access. Objects held in Java `static` fields are migrated to the more precious and expensive session space.

1.2.4 Footprint

The footprint of a running Java program is affected by many factors:

- Size of the program

  The size of the program depends on the number of classes and methods and how much code they contain.

- Complexity of the program

  The complexity of the program depends on the number of core class libraries that the Oracle JVM uses as the program runs, as opposed to the program itself.

- Amount of space the JVM uses

  The amount of space the JVM uses depends on the number of objects the JVM allocates, how large these objects are, and how many objects must be retained across calls.

- Ability of the garbage collector and memory manager to deal with the demands of the program running

  This can not be determined often. The speed with which objects are allocated and the way they are held on to by other objects influences the importance of this factor.

From a scalability perspective, the key to supporting multiple clients concurrently is a minimum per-user session footprint. The Oracle JVM keeps the per-user session footprint to a minimum by placing all read-only data for users, such as Java bytecodes, in shared memory. Appropriate garbage collection algorithms are applied against call and session memories to maintain a small footprint for the user's session. The Oracle JVM uses the following types of garbage collection algorithms to maintain the user's session memory:

- Generational scavenging for short-lived objects
- Mark and lazy sweep collection for objects that exist for the life of a single call
- Copying collector for long-lived objects, that is, objects that live across calls within a session

1.2.5 Performance of an Oracle JVM

The performance of an Oracle JVM is enhanced by the embedding of an innovative Just-In-Time compiler similar to HotSpot on standard JVM. The platform-independent Java bytecodes run on top of a JVM, and the JVM interacts with the specific hardware platform. Any time you add levels within software, the performance is degraded. Because Java requires going through an intermediary to interpret the bytecodes, a degree of inefficiency exists for Java applications as compared to applications developed using a platform-dependent language, such as C. To address this issue, several JVM suppliers create native compilers. Native compilers translate Java bytecodes into platform-dependent native code, which eliminates the interpreter step and improves performance.

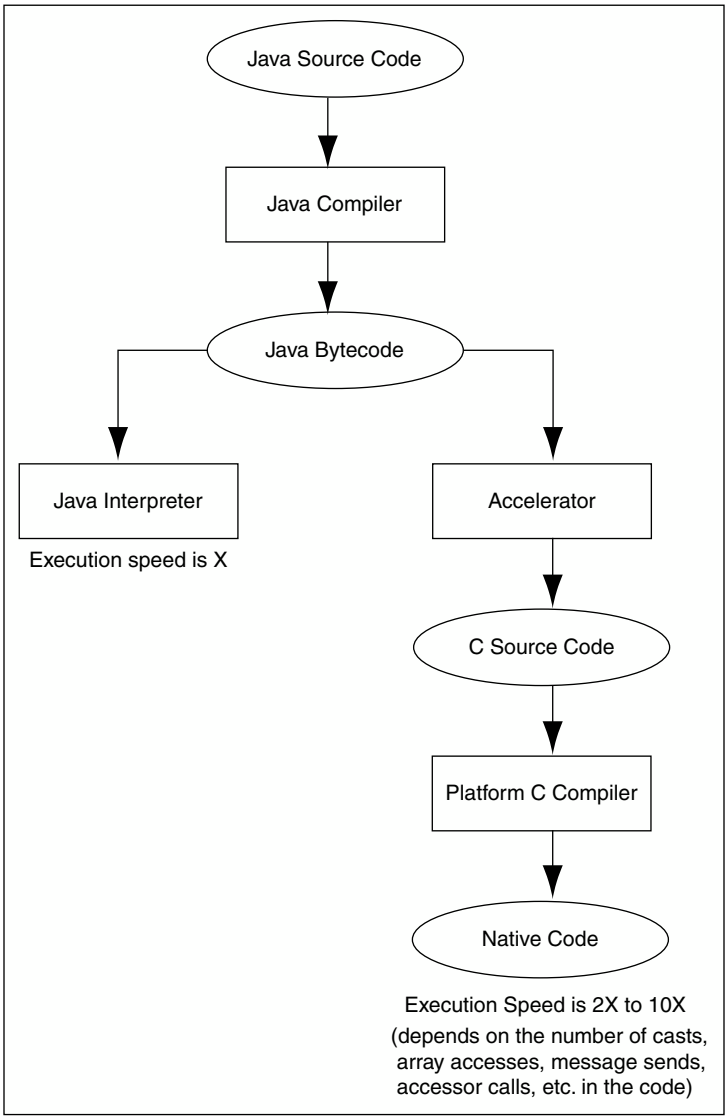
The following table describes two methods for native compilation:

Compiler	Description
Just-In-Time (JIT) Compilation	JIT compilers quickly compile Java bytecodes to platform-specific, or native, machine code during run time. These compilers do not produce an executable file to be run on the platform. Instead, they provide platform-dependent code from Java bytecodes that is run directly after it is translated. JIT compilers should be used for Java code that is run frequently and at speeds closer to that of code developed in other languages, such as C.
Ahead-of-Time Compilation	This compilation translates Java bytecodes to platform-independent C code before run time. Then a standard C compiler compiles the C code into an executable file for the target platform. This approach is more suitable for Java applications that are not modified frequently. This approach takes advantage of the mature and efficient platform-specific compilation technology found in modern C compilers.

Oracle Database uses Just-In-Time (JIT) compilation to deliver its core Java class libraries, such as JDBC code, in natively compiled form. The JIT compiler is enabled without the support of any plug-ins and it is applicable across all the platforms that Oracle supports.

The following figure illustrates how natively compiled code runs up to 10 times faster than interpreted code. As a result, the more native code your program uses, the faster it runs.

Figure 1-7 Interpreter versus Accelerator



Related Topics

- Oracle JVM Just-in-Time Compiler (JIT)

1.2.6 Dynamic Class Loading

Another strong feature of Java is dynamic class loading. The class loader loads classes from the disk and places them in the JVM-specific memory structures necessary for interpretation. The class loader locates the classes in `CLASSPATH` and loads them only when they are used

while the program is running. This approach, which works well for applets, poses the following problems in a server environment:

Problem	Description	Solution
Predictability	The class loading operation places a severe penalty when the program is run for the first time. A simple program can cause an Oracle JVM to load many core classes to support its needs. A programmer cannot easily predict or determine the number of classes loaded.	The Oracle JVM loads classes dynamically, just as with any other JVM. The same one-time class loading speed hit is encountered. However, because the classes are loaded into shared memory, no other users of those classes will cause the classes to load again, and they will use the same preloaded classes.
Reliability	A benefit of dynamic class loading is that it supports program updating. For example, you would update classes on a server, and clients, who download the program and load it dynamically, see the update whenever they next use the program. Server programs tend to emphasize reliability. As a developer, you must know that every client runs a specific program configuration. You do not want clients to inadvertently load some classes that you did not intend them to load.	Oracle Database separates the upload and resolve operation from the class loading operation at run time. You upload Java code you developed to the server using the `loadjava` tool. Instead of using `CLASSPATH`, you specify a resolver at installation time. The resolver is analogous to `CLASSPATH`, but enables you to specify the schemas in which the classes reside. This separation of resolution from class loading ensures that you always know what programs users run.

1.3 Overview of Oracle JVM

The Oracle JVM is a standard, Java-compatible environment that runs any pure Java application. It is compatible with the standard JLS and the JVM specifications. It supports the standard Java binary format and the standard Java APIs. In addition, Oracle Database adheres to standard Java language semantics, including dynamic class loading at run time.

Java in Oracle Database introduces the following terms:

- Session

  A session in Oracle Database Java environment is identical to the standard Oracle Database usage. A session is typically, although not necessarily, bounded by the time a single user connects to the server. As a user who calls a Java code, you must establish a session in the server.

- Call

  When a user causes a Java code to run within a session, it is termed as a call. A call can be started in the following different ways:

  – A SQL client program runs a Java stored procedure.

  – A trigger runs a Java stored procedure.

  – A PL/SQL program calls a Java code.

  In all the cases defined, a call begins, some combination of Java, SQL, or PL/SQL code is run to completion, and the call ends.

> **Note:**
>
> The concept of session and call apply across all uses of Oracle Database.

Unlike other Java environments, the Oracle JVM is embedded within Oracle Database and introduces a number of new concepts. This section discusses some important differences between an Oracle JVM and typical client JVMs based on:

- Process Area
- Java session initialization_ duration and entrypoints
- The GUI
- The IDE

1.3.1 Process Area

In a standard Java environment, you run a Java application through the interpreter by issuing the following command on the command line, where *classname* is the name of the class that you want the JVM to interpret first:

```
java classname
```

When using the Oracle JVM, you must load the application into the database, publish the interface, and then run the application within a database session. The database session is the environment in which the Oracle JVM runs and as such is the analog of the operating system process in which a standard client JVM runs.

Related Topics

- Java Applications on Oracle Database

1.3.2 Java session initialization, duration and entrypoints

Standard client-based Java applications declare a single, top-level method, `public static void main(String args[])`. This method is executed once and the instantiation of the Java Virtual Machine lasts for the duration of that call. But, Oracle Java applications are not restricted to a single top-level main entrypoint, and the duration of the Oracle JVM instantiation is not determined by a single call and the exit of the call from this entrypoint. Each client begins a session, calls its server-side logic modules through top-level entry points, and eventually ends the session. The same JVM instance remains in place for the entire duration of the session, so data state such as static variable values can be used across multiple calls to multiple top-level entry points.

Class definitions that have been loaded and published in the database are generally available to all sessions in that database. The JVM instance in a given session and the Java data objects and global field values in that JVM instance are private to the session. This data is present for the duration of the session and may be used by multiple calls within the lifetime of that session. But, neither this data is visible to other sessions nor the data can be shared in any way with other sessions. This is analogous to how in a standard client Java application separate invocations of the main method share the same class definitions, but the data created and used during those invocations are separate.

1.3.2.1 Support for JDK 11 and Modules

Starting from this release, Oracle JVM supports JDK 11 and Java modules.

In JDK 11, you can associate a client-based Java application with one or more modules. The main Java class can be a member of a module, and the module can be added to the application through the Java command line argument `--add-modules`. These modules can, in turn, require other modules. During VM initialization, this set of modules is examined for consistency and completeness. All the modules required by the application, at any time, must be a part of the initial set of modules, which was included at VM start up.

Oracle JVM in JDK 11 functions in a similar manner. All `java.se` modules are automatically included in the root set of modules. The `main` class of an Oracle JVM session is the class of the method, whose Java stored procedure is invoked by the database call that initiates the Oracle JVM session. If this class is a member of a module, then it is added to the module root set. Other modules can be added using one of the following ways:

- By specifying the `loadjava --add-modules` option, when the `main` class is loaded already
- By specifying the `oracle.aurora.addmods` system property

The `oracle.aurora.addmods` system property has the same affect as specifying the client-based `java --add-modules` command-line argument. Alternatively, if the `main` class was loaded with the `loadjava -add-modules` command-line argument, then you do not need to specify the user of the `oracle.aurora.addmods` property.

In client-side Java, any modules, explicitly required by the above modules, are also included. As mentioned earlier, the final set of modules is examined for consistency and completeness at Oracle JVM session start up. In client-side Java, all modules required at any time by the Oracle Java session, including any modules called during any subsequent database calls in that session, must be members of the initial set of modules, which was included at Oracle JVM session start up.

1.3.3 The GUI

A server cannot provide GUIs, but it can provide the logic that drives them. The Oracle JVM supports only the headless mode of the Java Abstract Window Toolkit (AWT). All Java AWT classes are available within the server environment and your programs can use the Java AWT functionality, as long as they do not attempt to materialize a GUI on the server.

Related Topics

- User Interfaces on the Server

1.3.4 The IDE

The Oracle JVM is oriented to Java application deployment, and not development. You can write and test applications on any preferred integrated development environment (IDE), such as Oracle JDeveloper, and then deploy them within the database for the clients to access and run them.

> **See Also:**
>
> "Development Tools"

The binary compatibility of Java enables you to work on any IDE and then upload the Java class files to the server. You need not move your Java source files to the database. Instead, you can use powerful client-side IDEs to maintain Java applications that are deployed on the server.

Related Topics

- Development Tools

1.4 Feature List of Oracle JVM

This section lists the features of Oracle JVM and the versions in which they were first supported.

Table 1-1 Feature List of Oracle JVM

Feature	Supported Since Oracle JVM Release
`loadjava` URL support	11.1
List-Based operations with `dropjava` support	11.1
`ojvmtc` Tool	11.1
`Runjava` command-line interface (JDK-like interface)	11.1
Database-Resident JARs	11.1
Sharing of user classloaded classes metadata support	11.1
Two-tier duration for the Java session state support	11.1
Default service feature	11.1
Just-in-Time compiler (JIT)	11.1
Internet Protocol Version 6 (IPv6) Support	11.2

1.5 Main Components of Oracle JVM

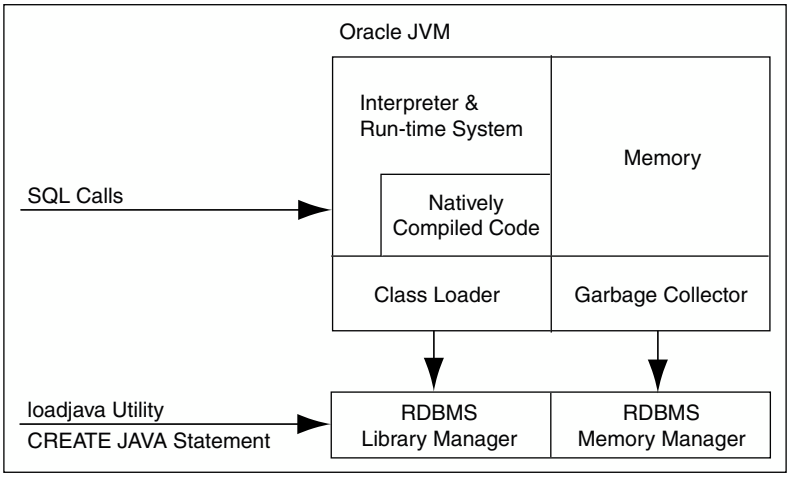
This section briefly describes the main components of an Oracle JVM and some of the facilities they provide.

The Oracle JVM is a complete, Java 2-compliant environment for running Java applications. It runs in the same process space and address space as the database kernel by sharing its memory heaps and directly accessing its relational data. This design optimizes memory use and increases throughput.

The Oracle JVM provides a run-time environment for Java objects. It fully supports Java data structures, method dispatch, exception handling, and language-level threads. It also supports all the core Java class libraries, including `java.lang`, `java.io`, `java.net`, `java.math`, and `java.util`.

Figure 1-8 shows the main components of an Oracle JVM.

Figure 1-8 Main Components of an Oracle JVM



The Oracle JVM embeds the standard Java namespace in the database schemas. This feature lets Java programs access Java objects stored in Oracle Database and application servers across the enterprise.

In addition, the Oracle JVM is tightly integrated with the scalable, shared memory architecture of the database. Java programs use call, session, and object lifetimes efficiently without user intervention. As a result, the Oracle JVM and middle-tier Java business objects can be scaled, even when they have session-long state.

The following sections provide an overview of some of the components of the Oracle JVM and the JDBC driver:

- Library Manager
- Compiler
- Interpreter
- Class Loader
- Verifier
- Server-Side JDBC Internal Driver
- System Classes

1.5.1 Library Manager

To store Java classes in Oracle Database, you use the `loadjava` command-line tool, which uses the SQL `CREATE JAVA` statements to do its work. When called by the `CREATE JAVA {SOURCE | CLASS | RESOURCE}` statement, the library manager loads Java source, class, or resource files into the database. These Java schema objects are not accessed directly, and only an Oracle JVM uses them.

1.5.2 Compiler

The Oracle JVM includes a standard Java compiler. When the `CREATE JAVA SOURCE` statement is run, it translates Java source files into architecture-neutral, one-byte instructions known as bytecodes. Each bytecode consists of an opcode followed by its operands. The resulting Java

class files, which conform fully to the Java standard, are submitted to the interpreter at run time.

1.5.3 Interpreter

To run Java programs, the Oracle JVM includes a standard Java 2 byte code interpreter. The interpreter and the associated Java run-time system run standard Java class files. The run-time system supports native methods and call-in and call-out from the host environment.

> **Note:**
>
> You can also compile your Java code to improve performance. The Oracle JVM uses natively compiled versions of the core Java class libraries and JDBC drivers.

1.5.4 Class Loader

In response to requests from the run-time system, the Java class loader locates, loads, and initializes Java classes stored in the database. The class loader reads the class and generates the data structures needed to run it. Immutable data and metadata are loaded into initialize-once shared memory. As a result, less memory is required for each session. The class loader attempts to resolve external references when necessary. In addition, if the source files are available, then the class loader calls the Java compiler automatically when Java class files must be recompiled.

1.5.5 Verifier

Java class files are fully portable and conform to a well-defined format. The verifier prevents the inadvertent use of spoofed Java class files, which might alter program flow or violate access restrictions. Oracle security and Java security work with the verifier to protect your applications and data.

1.5.6 Server-Side JDBC Internal Driver

JDBC is a standard and defines a set of Java classes providing vendor-independent access to relational data. The JDBC classes are modeled after ODBC and the X/Open SQL Call Level Interface (CLI) and provide standard features, such as simultaneous connections to several databases, transaction management, simple queries, calls to stored procedures, and streaming access to `LONG` column data.

Using low-level entry points, a specially tuned JDBC driver runs directly inside Oracle Database, providing fast access to Oracle data from Java stored procedures. The server-side JDBC internal driver complies fully with the standard JDBC specification. Tightly integrated with the database, the JDBC driver supports Oracle-specific data types, globalization character sets, and stored procedures. In addition, the client-side and server-side JDBC APIs are the same, which makes it easy to partition applications.

1.5.7 System Classes

A set of classes that constitute a significant portion of the implementation of Java in Oracle Database environment is known as the **System classes**. These classes are defined in the `SYS` schema and exported for all users by public synonym.

In JDK 11, all system classes are members of modules. Each module defines a set of packages. The module system does not allow individual packages to be *split* across modules, including across a named module and the *unnamed* module. Oracle JVM also does not support module patching. Therefore, redefining system classes is completely prohibited.

> **See Also:**
>
> Java Platform, Standard Edition & Java Development Kit Version 11 API Specification

A class with the same name as one of the System classes can be defined in a schema other than the `SYS` schema[1]. But, this is a bad practice because the alternate version of the class may behave in a manner that violates assumptions about the semantics of that class that are present in other System classes or in the underlying implementation of Java Virtual Machine. Oracle strongly discourages this practice.

1.6 Java Programming in Oracle Database

Oracle provides enterprise application developers an end-to-end Java solution for creating, deploying, and managing Java applications. The total solution consists of client-side and server-side programmatic interfaces, tools to support Java development, and a JVM integrated with Oracle Database. All these products are fully compatible with Java standards. This section covers the following topics:

- Java in Database Application Development
- Java Programming Environment Usage
- Java Stored Procedures
- PL/SQL Integration and Oracle RDBMS Functionality
- Development Tools
- Internet Protocol Version 6 Support

1.6.1 Java in Database Application Development

The most important features of Java in database application development are:

- Designing data-bound procedures and functions using Java SE APIs and JDBC.
- Extending the reach and capabilities of the database with standard and third-party Java libraries. For example, accessing third-party databases using their drivers in the database and accessing Hadoop/HDFS.
- Providing flexible partitioning of Java2 Platform, Standard Edition (J2SE) applications for symmetric data access at the JDBC level.
- Bridging SQL and the Java2 Platform, Enterprise Edition (J2EE) world by:
  - Calling out Web components, such as JSP and servlet
  - Bridging SQL and Web Services using Web Service Callouts

[1] You cannot always define a class with the same name as one of the System classes. For the classes present in some packages, for example, `java.lang`, such definitions are explicitly prohibited by the code.

- Using an Oracle JVM as ERP Integration Hub.
- Invalidating cache.

1.6.2 Java Programming Environment Usage

In addition to the Oracle JVM, the Java programming environment provides:

- Java stored procedures as the Java equivalent and companion for PL/SQL. Java stored procedures are tightly integrated with PL/SQL. You can call Java stored procedures from PL/SQL packages and PL/SQL procedures from Java stored procedures.
- The JDBC programming interfaces for accessing SQL data.
- Tools and scripts that assist in developing, loading, and managing classes.

The following table helps you decide when to use which Java API:

Type of functionality you need	Java API to use
To have a Java procedure called from SQL, such as a trigger.	Java stored procedures
To call dynamic, complex SQL statements from a Java object.	JDBC

1.6.3 Java Stored Procedures

Java stored procedures are Java programs written and deployed on a server and run from the server, exactly like a PL/SQL stored procedure. You invoke it directly with products like SQL*Plus, or indirectly with a trigger. You can access it from any Oracle Net client, such as OCI and PRO*, or JDBC.

In addition, you can use Java to develop powerful, server-side programs, which can be independent of PL/SQL. Oracle Database provides a complete implementation of the standard Java programming language and a fully compliant JVM.

Related Topics

- Developing Java Stored Procedures

1.6.4 PL/SQL Integration and Oracle RDBMS Functionality

You can call existing PL/SQL programs from Java and Java programs from PL/SQL. This solution protects and leverages your PL/SQL and Java code and opens up the advantages and opportunities of Java-based Internet computing.

Oracle Database offers JDBC Java APIs for accessing SQL data, which are available on the client, and also on the server. As a result, you can deploy your applications on both the client and the server.

1.6.4.1 JDBC Drivers

JDBC is a database access protocol that enables you to connect to a database and run SQL statements and queries to the database. The core Java class libraries provide the following JDBC APIs: `java.sql` and `javax.sql`. However, JDBC is designed to enable vendors to supply drivers that offer the necessary specialization for a particular database. Oracle provides the following distinct JDBC drivers:

Driver	Description
JDBC Thin driver	You can use the JDBC Thin driver to write pure Java applications and applets that access Oracle SQL data. The JDBC Thin driver is especially well-suited for Web-based applications and applets, because you can dynamically download it from a Web page, similar to any other Java applet.
JDBC OCI driver	The JDBC OCI driver accesses Oracle-specific native code, that is, non-Java code, and libraries on the client or middle tier, providing performance boost compared to the JDBC Thin driver, at the cost of significantly larger size and client-side installation.
JDBC server-side internal driver	Oracle Database uses the server-side internal driver when the Java code runs on the server. It allows Java applications running in the Oracle JVM on the server to access locally defined data, that is, data on the same system and in the same process, with JDBC. It provides a performance boost, because of its ability to use the underlying Oracle RDBMS libraries directly, without the overhead of an intervening network connection between the Java code and SQL data. By supporting the same Java-SQL interface on the server, Oracle Database does not require you to rework code when deploying it.

Related Topics

- About Utilizing JDBC with Java in the Database

See Also:

Oracle Database JDBC Developer's Guide

1.6.5 Development Tools

The introduction of Java in Oracle Database enables you to use several Java IDEs. The adherence of Oracle Database to the Java standards and specifications and the open Internet standards and protocols ensures that your Java programs work successfully, when you deploy them on Oracle Database. Oracle provides many tools or utilities that are written in Java making development and deployment of Java server applications easier. Oracle JDeveloper, a Java IDE provided by Oracle, has many features designed specifically to make deployment of Java stored procedures and EJBs easier. You can download JDeveloper from

`http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html`

1.6.6 Internet Protocol Version 6 Support

Oracle JVM supports Internet Protocol Version 6 (IPv6) addresses in the URL and machine names of the Java code in the database, which resolve to IPv6 addresses.

The primary benefit of IPv6 is a large address space, derived from the use of 128-bit addresses. IPv6 also improves upon IPv4 in areas such as routing, network autoconfiguration, security, quality of service, and so on.

The following system properties enable you to configure IPv6 preferences:

java.net.preferIPv4Stack

If IPv6 is available on the operating system, then the underlying native socket will be an IPv6 socket. This enables Java applications to connect to, and accept connections from both IPv4 and IPv6 hosts. If you have an application that has a preference to use only IPv4 sockets, then you can set this property to `true`. If you set the property to `true`, then it implies that the application will not be able to communicate with IPv6 hosts.

java.net.preferIPv6Addresses

Even if IPv6 is available on the operating system, then for backward compatibility reasons, the addresses are mapped to IPv4. For example, applications that depend on access to only an IPv4 service, benefit from this type of mapping. If you want to change the preferences to use IPv6 addresses over IPv4 addresses, then you can set the `java.net.preferIPv6Addresses` property to `true`. This allows applications to be tested and deployed in environments, where the application is expected to connect to IPv6 services.

> **Note:**
>
> All the new System classes that are required for IPv6 support are loaded when Java is enabled during database initialization. So, if your application does not have any addresses that are included in the software code, then you do not need to change your code to use IPv6 functionality.

1.7 Memory Model for Dedicated Mode Sessions

Since Oracle Database 10*g*, the Oracle JVM has a new memory model for sessions that connect to the database through a dedicated server. The basic memory structures associated with Oracle include:

- System Global Area (SGA)

  The SGA is a group of shared memory structures, known as SGA components, that contain data and control information for one Oracle Database instance. The SGA is shared by all server and background processes. Examples of data stored in the SGA include cached data blocks and shared SQL areas.

- Program Global Areas (PGA)

  A PGA is a memory region that contains data and control information for a server process. It is nonshared memory created by Oracle when a server process is started. Access to the PGA is exclusive to the server process. There is one PGA for each server process. Background processes also allocate their own PGAs. The total PGA memory allocated for all background and server processes attached to an Oracle instance is referred to as the **aggregate PGA**.

The simplest way to manage memory is to allow the database to automatically manage and tune it for you. To do so, you set only a target memory size initialization parameter (`MEMORY_TARGET`) and a maximum memory size initialization parameter (`MEMORY_MAX_TARGET`), on most platforms. The database then tunes to the target memory size, redistributing memory as needed between the SGA and aggregate PGA. Because the target memory initialization parameter is dynamic, you can change the target memory size at any time without restarting the database. The maximum memory size serves as an upper limit so that you cannot accidentally set the target memory size too high. Because certain SGA components either

cannot easily shrink or must remain at a minimum size, the database also prevents you from setting the target memory size too low.

> **See Also:**
>
> *Oracle Database Administrator's Guide*