2

Java Applications on Oracle Database

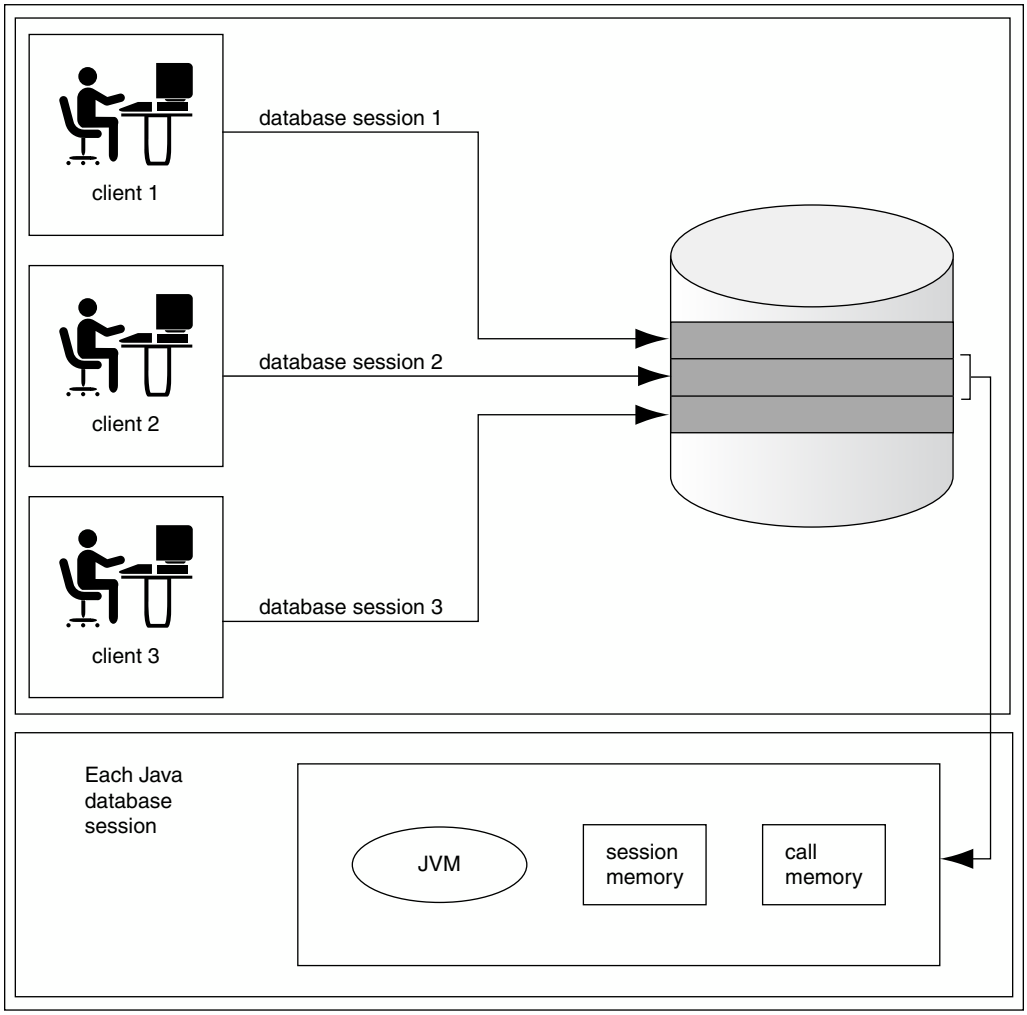
Oracle Database runs standard Java applications. However, the Java-integrated Oracle Database environment is different from a typical Java development environment. This chapter describes the basic differences for writing, installing, and deploying Java applications within Oracle Database in the following sections:

2.1 Database Sessions Imposed on Java Applications

In the Java-integrated Oracle Database, your Java applications exist within the context of a database session. Oracle JVM sessions are entirely analogous to traditional Oracle sessions. Each Oracle JVM session maintains the state of the Java applications accessed by the client across calls within the session.

Figure 2-1 illustrates how each Java client starts a database session as the environment for running Java applications within the database. Each Java database session has a separate garbage collector, session memory, and call memory.

Figure 2-1 Java Environment Within Each Database Session



Within the context of a session, the client performs the following:

1. Connects to the database and opens a session.
2. Runs Java within the database. This is referred to as a call. The first call within the database session, which uses Java, starts the Oracle JVM session. This call also initializes the module system.

> **See Also:**
>
> Java session initialization, duration and entrypoints

3. Continues to work within the session, performing as many calls as required.
4. Ends the session.

Within a session, the client has its own Java environment. It appears to the client as if a separate, individual JVM was started for each session, although the implementation is more efficient than this seems to imply. Within a session, Oracle JVM manages the scalability of applications. Every call from a single client is managed within its own session, and calls from each client is handled separately. Oracle JVM maximizes sharing read-only data between

clients and emphasizes a minimum amount of per-session incremental footprint, to maximize performance for multiple clients.

The underlying server environment hides the details associated with session, network, state, and other shared resource management issues from the Java code. Fields defined as `static` are local to the client. No client can access the `static` fields of other clients, because the memory is not available across session boundaries. Because each client runs the Java application calls within its own session, activities of each client are separate from any other client. During a call, you can store objects in `static` fields of different classes, which will be available in the next call. The entire state of your Java program is private and exists for your entire session.

Oracle JVM manages the following within the session:

- All the objects referenced by `static` Java fields, all the objects referred to by these objects, and so on, till their transitive closure
- Garbage collection for the client that created the session
- Session memory for `static` fields and across call memory needs
- Call memory for fields that exist within a call

2.2 Execution Control of Java Applications

In the Java 2 Platform, Standard Edition (J2SE) environment, you develop Java applications with a `main()` method, which is called by the interpreter when the class is run. The `main()` method is called when you enter the following command on the command-line:

```
java classname
```

This command starts the Java interpreter and passes the desired class, that is, the class specified by *`classname`*, to the Java interpreter. The interpreter loads the class and starts running the application by calling `main()`. However, Java applications within the database do not start by a call to the `main()` method.

After loading your Java application within the database, you can run it by calling any `static` method within the loaded class. The class or methods must be published before you can run them. In Oracle Database, the entry point for Java applications is not assumed to be `main()`. Instead, when you run your Java application, you specify a method name within the loaded class as your entry point.

For example, in a standard Java environment, you would start the Java object on the server by running the following command:

```
java myprogram
```

where, *`myprogram`* is the name of a class that contains the `main()` method. In *`myprogram`*, `main()` immediately calls `mymethod()` for processing incoming information.

In Oracle Database, you load the `myprogram.class` file into the database and publish `mymethod()` as an entry-point. Then, the client or trigger can invoke `mymethod()` explicitly.

2.3 Java Code, Binaries, and Resources Storage

In the standard Java development environment, Java source code, binaries, and resources are stored as files in a file system, as follows:

- Source code files are saved as `.java` files.
- Compiled Java binary files are saved as `.class` files.
- Resources are any data files, such as `.properties` files that are stored in the file system hierarchy, which are loaded and used at run time.

In addition, when you run a Java application, you specify the `CLASSPATH`, which is a file or directory path in the file system that contains your `.class` files. Java also provides a way to group these files into a single archive form, a ZIP or Java Archive (JAR) file.

Both these concepts are different in Oracle Database environment.

Table 2-1 describes how Oracle Database handles Java classes and locates dependent classes.

Table 2-1 Description of Java Code and Classes Storage in Oracle Database

Tasks	How it differs for Oracle JVM
Storing Java code, binaries, and resources	In Oracle Database, source code, classes, and resources reside within the database and are known as Java schema objects, where a schema corresponds to a database user. There are three types of Java schema objects: source, class, and resource. There are no `.java`, `.class`, `.properties` files on the server. Instead, these files map to the appropriate Java schema objects.
Locating Java classes	Instead of the `CLASSPATH`, you use a resolver to specify one or more schemas to search for Java source, class, and resource schema objects.

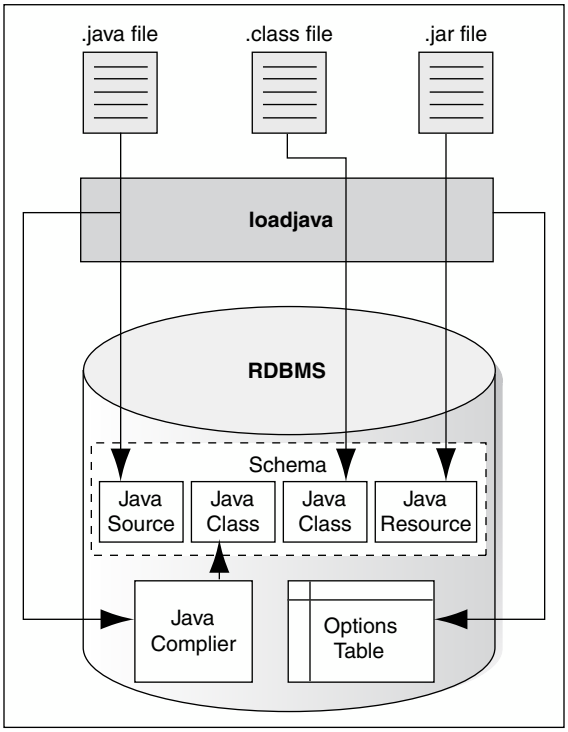
2.4 About Java Classes Loaded in the Database

If you are not using the command-line interface, then to make the Java files available to Oracle JVM, you must load them into the Database as schema objects.

If you are not using the command-line interface, you must load Java files into the database as schema objects, to make them available to Oracle JVM. The `loadjava` tool can call the Java compiler of Oracle JVM, which compiles source files into standard class files.

The following figure shows that the `loadjava` tool can set the values of options stored in a system database table. Among other things, these options affect the processing of Java source files.

Figure 2-2 Loading Java into Oracle Database



Each Java class is stored as a schema object. The name of the object is derived from the fully qualified name of the class, which includes the names of containing packages. For example, the full name of the class `Handle` is:

```
oracle.aurora.rdbms.Handle
```

In the Java schema object name, slashes replace periods, so the full name of the class becomes:

```
oracle/aurora/rdbms/Handle
```

Starting with JDK 11, the names of the Java schema objects that are members of modules, are of the following form:

```
<module_name>///<class_name>
```

where, `<module_name>` is the actual module name, with no character replacement. For example, the Java schema object name of the `oracle.aurora.rdbms.Handle` class, which is a member of the `oracle.aurora` module, is `oracle.aurora///oracle/aurora/rdbms/Handle`.

Oracle Database accepts Java names up to 4000 characters long. However, the names of Java schema objects cannot be longer than 128 characters. Therefore, if a schema object name is longer than 128 characters, then the system generates a short name, or alias, for the schema object. Otherwise, the fully qualified name, also called full name, is used. You can specify the full name in any context that requires it. When needed, name mapping is handled by Oracle Database.

Related Topics

- About Using the Command-Line Interface

- Shortened Class Names
  Each Java source, class, and resource is stored in its own schema object in the server. The name of the schema object is derived from the fully qualified name, which includes relevant path or package information. Dots are replaced by slashes.
- System Classes

2.5 Preparing Java Class Methods for Execution

To ensure that your Java methods run, you must do the following:

1. Decide when the Java source code is going to be compiled.
2. Decide if you are going to use the default resolver or another resolver for locating supporting Java classes within the database.
3. Load the classes into the database. If you do not wish to use the default resolver for your classes, then you should specify a separate resolver with the load command.
4. Publish your class or method.

This sections covers the following topics:

- Compiling Java Classes
- Overview of Resolving Class Dependencies
- Overview of Loading Classes Using the loadjava Tool
- Overview of Granting Execute Rights
- Overview of Controlling the Current User
- Overview of Checking Java Uploads
- About Publishing Java Methods Loaded in the Database
- Overview of Auditing Java Classes Loaded in the Database

2.5.1 Compiling Java Classes

Compilation of the Java source code can be done in one of the following ways:

- You can compile the source explicitly on a client system before loading it into the database, through a Java compiler, such as `javac`.
- You can ask the database to compile the source during the loading process, which is managed by the `loadjava` tool.
- You can force the compilation to occur dynamically at run time.

> **Note:**
>
> If you decide to compile through the `loadjava` tool, then you can specify the compiler options.

This section includes the following topics:

- Compiling Source Through javac
- Compiling Source Through the loadjava Tool

- Compiling Source at Run Time
- Specifying Compiler Options
- Recompiling Source Programs Automatically

Related Topics

- Specifying Compiler Options

2.5.1.1 Compiling Source Through javac

You can compile Java source code with a conventional Java compiler as shown in the following example:

> **Note:**
>
> You must ensure that you are using a JDK version that is compatible with what Oracle JVM supports. For example, for the current release, it is JDK 11.

```
javac <file_name>.java
```

After compilation, you load the compiled binary into the database, rather than the source itself. This is a better option, because it is usually easier to debug the Java code on your own system, rather than debugging it on the database.

2.5.1.2 Compiling Source Through the loadjava Tool

When you specify the `-resolve` option with the `loadjava` tool for a source file, the following occurs:

1. The source file is loaded as a source schema object.
2. The source file is compiled.
3. Class schema objects are created for each class defined in the compiled `.java` file.
4. The compiled code is stored in the class schema objects.

Oracle Database writes all compilation errors to the log file of the `loadjava` tool as well as the `USER_ERRORS` view.

2.5.1.3 Compiling Source at Run Time

When you load the Java source into the database without the `-resolve` option, for example:

```
loadjava <file_name>.java
```

Then, Oracle Database compiles the source automatically when the class is needed during run time. The source file is loaded into a source schema object. Oracle Database writes all compilation errors to the log file of the `loadjava` tool as well as the `USER_ERRORS` view.

2.5.1.4 Specifying Compiler Options

You can specify the compiler options in the following ways:

- Specify compiler options on the command line with the `loadjava` tool. You can also specify the encoding option with the `loadjava` tool.

- Specify persistent compiler options in the `JAVA$OPTIONS` table. The `JAVA$OPTIONS` table exists for each schema. Every time you compile, the compiler uses these options. However, any compiler options specified with the `loadjava` tool override the options defined in this table. You must create this table yourself if you wish to specify compiler options in this manner.

2.5.1.4.1 Specifying Default Compiler Options

When compiling a source schema object for which neither a `JAVA$OPTIONS` entry exists nor a command-line value for any option is specified, the compiler assumes a default value as follows:

- `encoding=System.getProperty("file.encoding");`
- `debug=true`

  This option is equivalent to:

  ```
  javac -g
  ```

2.5.1.4.2 Specifying Compiler Options on the Command Line

The `encoding` compiler option specified with the `loadjava` tool identifies the encoding of the `.java` file. This option overrides any matching value in the `JAVA$OPTIONS` table. The values are identical to:

```
javac -encoding
```

This option is relevant only when loading a source file.

2.5.1.4.3 Specifying Compiler Options Specified in a Database Table

Each `JAVA$OPTIONS` entry contains the names of source schema objects to which an option setting applies. You can use multiple rows to set the options differently for different source schema objects.

You can set `JAVA$OPTIONS` entries by using the following procedures and functions, which are defined in the database package `DBMS_JAVA`:

```
PROCEDURE set_compiler_option(name VARCHAR2, option VARCHAR2, value VARCHAR2);

FUNCTION get_compiler_option(name VARCHAR2, option VARCHAR2) RETURNS VARCHAR2;

PROCEDURE reset_compiler_option(name VARCHAR2, option VARCHAR2);
```

2.5.1.4.4 Details About Specifying Compiler Options Specified in the Database Table

The following table describes the parameters for the methods described in the preceding section.

Table 2-2 Definitions for the Name and Option Parameters

Parameter	Description
name	This is a Java package name, a fully qualified class name, or an empty string. When the compiler searches the `JAVA$OPTIONS` table for the options to use for compiling a Java source schema object, it uses the row that has a value for `name` that most closely matches the fully qualified class name of a schema object. A `name` whose value is the empty string matches any schema object name.
option	The `option` parameter is either `online`, `encoding`, or `debug`.

Initially, a schema does not have a `JAVA$OPTIONS` table. To create a `JAVA$OPTIONS` table, use the `java.set_compiler_option` procedure from the `DBMS_JAVA` package to set a value. The procedure will create the table, if it does not exist. Specify parameters in single quotes. For example:

```
SQL> execute dbms_java.set_compiler_option('x.y', 'online', 'false');
```

The following table represents a hypothetical `JAVA$OPTIONS` database table. The pattern match rule is to match as much of the schema name against the table entry as possible. The schema name with a higher resolution for the pattern match is the entry that applies. Because the table has no entry for the `encoding` option, the compiler uses the default or the value specified on the command line. The `online` option shown in the table matches schema object names as follows:

- The name `a.b.c.d` matches class and package names beginning with `a.b.c.d`. The packages and classes are compiled with `online=true`.
- The name `a.b` matches class and package names beginning with `a.b`. The name `a.b` does not match `a.b.c.d`. The packages and classes are compiled with `online=false`.
- All other packages and classes match the empty string entry and are compiled with `online=true`.

Table 2-3 Example JAVA$OPTIONS Table

Name	Option	Value	Match Examples
a.b.c.d	`online`	`true`	• `a.b.c.d` Matches the pattern exactly. • `a.b.c.d.e` First part matches the pattern exactly. No other rule matches the full qualified name.
a.b	`online`	`false`	• `a.b` Matches the pattern exactly • `a.b.c.x` First part matches the pattern exactly. No other rule matches beyond this rule.

Table 2-3 (Cont.) Example JAVA$OPTIONS Table

Name	Option	Value	Match Examples
Empty string	`online`	`true`	• `a.c` No pattern match with any defined name. Defaults to the empty string rule. • `x.y` No pattern match with any defined name. Defaults to the empty string rule.

2.5.1.5 Recompiling Source Programs Automatically

Oracle Database provides a dependency management and automatic build facility that transparently recompiles source programs when you make changes to the source or binary programs upon which they depend. Consider the following example:

```
public class A
{
  B b;
  public void assignB()
  {
    b = new B()
  }
}
public class B
{
  C c;
  public void assignC()
  {
    c = new C()
  }
}
public class C
{
  A a;
  public void assignA()
  {
    a = new A()
  }
}
```

The system tracks dependencies at a class level of granularity. In the preceding example, you can see that classes `A`, `B`, and `C` depend on one another, because `A` holds an instance of `B`, `B` holds an instance of `C`, and `C` holds an instance of `A`. If you change the definition of class `A` by adding a new field to it, then the dependency mechanism in Oracle Database flags classes `B` and `C` as invalid. Before you use any of these classes again, Oracle Database attempts to resolve them and recompile, if necessary. Note that classes can be recompiled only if the source file is present on the server.

The dependency system enables you to rely on Oracle Database to manage dependencies between classes, to recompile, and to resolve automatically. You must force compilation and resolution yourself only if you are developing and you want to find problems early. The `loadjava` tool also provides the facilities for forcing compilation and resolution if you do not want the dependency management facilities to perform this for you.

2.5.2 Overview of Resolving Class Dependencies

Many Java classes contain references to other classes, which is the essence of reusing code. A conventional JVM searches for `.class`, `.zip`, and `.jar` files within the directories specified in `CLASSPATH`.

In contrast, Oracle JVM searches database schemas for class objects. In Oracle Database, because you load all Java classes into the database, you may need to specify where to find the dependent classes for your Java class within the database.

All classes loaded within the database are referred to as class schema objects and are loaded within certain schemas. All predefined Java application programming interfaces (APIs), such as `java.lang.*` , are loaded within the `PUBLIC` schema. If your classes depend on other classes you have defined, then you will probably load them all within your own schema. For example, if your schema is `HR`, then the database resolver searches the `HR` schema before searching the `PUBLIC` schema. The listing of schemas to search is known as a **resolver specification**. Resolver specifications are defined for each class. This is in contrast to a classic JVM, where `CLASSPATH` is global to all classes.

When locating and resolving the interclass dependencies for classes, the resolver marks each class as valid or invalid, depending on whether all interdependent classes are located. If the class that you load contains a reference to a class that is not found within the appropriate schemas, then the class is listed as invalid. Unsuccessful resolution at run time produces a `ClassNotFound` exception. Also, run-time resolution can fail for lack of database resources, if the tree of classes is very large.

> **Note:**
>
> As with the Java compiler, the `loadjava` tool resolves references to classes, but not to resources. Ensure that you correctly load the resource files that your classes require.

For each interclass reference in a class, the resolver searches the schemas specified by the resolver specification for a valid class schema object that satisfies the reference. If all references are resolved, then the resolver marks the class valid. A class that has never been resolved, or has been resolved unsuccessfully, is marked invalid. A class that depends on a schema object that becomes invalid is also marked invalid.

To make searching for dependent classes easier, Oracle Database provides a default resolver and resolver specification that searches the definer's schema first and then searches the `PUBLIC` schema. This covers most of the classes loaded within the database. However, if you are accessing classes within a schema other than your own or `PUBLIC`, you must define your own resolver specification.

Classes can be resolved in the following ways:

- Loading using the default resolver, which searches the definer's schema and `PUBLIC`:

  ```
  loadjava -resolve
  ```

- Loading using your own resolver specification definition:

  ```
  loadjava-resolve -resolver "((* HR)(* OTHER)(* PUBLIC))"
  ```

In the preceding example, the resolver specification definition includes the `HR` schema, `OTHER` schema, and `PUBLIC`.

The `-resolver` option specifies the objects to search within the schemas defined. In the preceding example, all class schema objects are searched within `HR`, `OTHER`, and `PUBLIC`. However, if you want to search for only a certain class or group of classes within the schema, then you could narrow the scope for the search. For example, to search only for the `my/gui/*` classes within the `OTHER` schema, you would define the resolver specification as follows:

```
loadjava -resolve -resolver '((* HR) ("my/gui/*" OTHER) (* PUBLIC))'
```

The first parameter within the resolver specification is for the class schema object, and the second parameter defines the schema within which to search for these class schema objects.

Starting with Oracle Database Release 23ai, Oracle JVM supports JDK 11, which has extended the format of resolver specifications to add module resolution information.

> **See Also:**
>
> The loadjava Tool

2.5.2.1 Allowing References to Nonexistent Classes

You can specify a special option within a resolver specification that allows an unresolved reference to a nonexistent class. Sometimes, internal classes are never used within a product.

In a standard Java environment, this is not a problem, because as long as the methods are not called, JVM ignores them. However, when resolving a class, Oracle JVM tries to resolve all names referenced by that class, including names that may never be used. If Oracle JVM cannot find a matching class for each such names referenced by that class, then the class being resolved is marked as invalid and cannot be run.

To ignore references, you can specify the wild card, minus sign (–), within the resolver specification. The following example specifies that any references to classes within `my/gui` are to be allowed, even if it is not present within the resolver specification schema list.

```
loadjava -resolve -resolver '((* HR) (* PUBLIC) ("my/gui/*" -))'
```

Without the wild card, if a dependent class is not found within one of the schemas, your class is listed as invalid and cannot be run.

In addition, you can define that all classes not found are to be ignored. However, this is dangerous, because a class that has a dependent class will be marked as valid, even if the dependent class does not exist. However, the class can never run without the dependent class. In this case, you will receive an exception at run time.

To ignore all classes not found within `HR` or `PUBLIC`, specify the following resolver specification:

```
loadjava -resolve -resolver "((* HR) (* PUBLIC) (* -))"
```

If you later intend to load the nonexistent classes that required you to use such a resolver, then you should not use a resolver containing the minus sign (-) wild card. Instead, include all referenced classes in the schema before resolving.

Even when a minus sign (-) wild card is used, the super class of a class can never be nonexistent. If the super class is not found, then the class will be invalid regardless of the use of a minus sign (-) wild card in the resolver.

> **Note:**
>
> For earlier releases, an alternative mechanism for dealing with the nonexistent classes is using the `-genmissing` option of the `loadjava` tool. This option causes the `loadjava` tool to create and load definitions of classes that are referenced, but not defined.
>
> Starting from Oracle Database Release 23ai, the `-genmissing` option has been deprecated in favor of the `(* -)` resolver terms.

2.5.2.2 Bytecode Verifier

According to JVM specification, `.class` files are subject to verification before the class they define is available in a JVM. In Oracle JVM, the verification process occurs at class resolution.

The following table describes the problems the resolver may find and the appropriate Oracle error code issued.

Table 2-4 ORA Errors

Error Code	Description
`ORA-29545`	If the resolver determines that the class is malformed, then the resolver does not mark it valid. When the resolver rejects a class, it issues an `ORA-29545` error. The `loadjava` tool reports the error. For example, this error is thrown if the contents of a `.class` file are not the result of a Java compilation or if the file has been corrupted. The `ORA-29545` error may also show up if you used the minus sign (-) wild card expression with the resolver and the validity of some classes was not verified.
`ORA-29552`	In some situations, the resolver allows a class to be marked valid, but will replace bytecodes in the class to throw an exception at run time. In these cases, the resolver issues an `ORA-29552` warning that the `loadjava` tool reports. The `loadjava` tool issues this warning when the Java Language Specification (JLS) requires an `IncompatibleClassChangeError` to be thrown. Oracle JVM relies on the resolver to detect these situations, supporting the proper run-time behavior that the JLS requires.

A resolver with the minus sign (-) wildcard marks your class valid, regardless of whether classes referenced by your class are present. Because of inheritance and interfaces, you may want to write valid Java methods that use an instance of a class as if it were an instance of a superclass or of a specific interface. When the method being verified uses a reference to class `A` as if it were a reference to class `B`, the resolver must check that `A` either extends or implements `B`. For example, consider the following potentially valid method, whose signature implies a return of an instance of `B`, but whose body returns an instance of `A`:

```
B myMethod(A a)
{
  return a;
}
```

The method is valid only if `A` extends the class `B` or `A` implements the interface `B`. If `A` or `B` have been resolved using the minus sign (-) wildcard, then the resolver does not know that this

method is safe. In this case, the resolver replaces the bytecodes of `myMethod` with bytecodes that throw an exception if `myMethod` is called.

A resolver without the minus sign (`-`) wildcard ensures that the class definitions of `A` and `B` are found and resolved properly if they are present in the schemas they specifically identify. The only time you may consider using the alternative resolver is if you must load an existing JAR file containing classes that reference other nonsystem classes, which are not included in the JAR file.

Related Topics

- Schema Objects and Oracle JVM Utilities

2.5.3 Logging in Oracle JVM

Oracle JVM extends the JDK Java Logging API in the area of logging properties lookup to enhance security of logging configuration management and to support logging configurations on a user basis.

> **See Also:**
>
> For more information about Java Logging APIs, visit the following site:
>
> `http://docs.oracle.com/javase/7/docs/`

You must activate the `LogManager` in the session to initialize the logging properties in Oracle JVM. The logging properties are initialized once per session with the `LogManager` API that is extended with the database resident resource lookup.

Oracle JVM performs the following steps to configure logging options:

1. If the `java.util.logging.config.class` property is set, then the logging behavior is the same as in standard JDK.
2. If the `java.util.logging.config.class` property is not set, then Oracle JVM inspects the availability of the `javavm/lib/logging.properties` resource in the current user schema.

   If available, this resource is used as the configuration setting for the `LogManager` and the `java.util.logging.config.file` property is set.
3. If both the above conditions do not hold true, then the `java.util.logging.config.file` property is inspected and if specified, it is used as described in `LogManager` API.
4. If none of the conditions in step 1, 2, and 3 holds true, then the `javavm/lib/logging.properties` resource in the `SYS` schema is used. This resource is a copy of the `$(java.home)/lib/logging.properties` file that is loaded into the `SYS` schema at database creation time. This means, by default, the `LogManager` behaves as if it is configured as per the `$(java.home)/javavm/lib/logging.properties` file. However, altering this file has no effect until the database is re-created

If you are not satisfied with the default settings in the `javavm/lib/logging.properties` file, then prepare a different set of properties and load them in your schema using the `loadjava` command. For example, if your schema is `HR` and your current file directory is `mydir`, then create a directory `javavm/lib/` under `mydir` and specify the required properties in the `logging.properties` file under the `mydir/javavm/lib/` directory. Then, invoke the `loadjava` command from `mydir` as follows:

```
mydir% loadjava -u HR -v -r javavm/lib/logging.properties
password:<password>
```

After invoking the `loadjava` command, you can delete the `mydir/javavm/lib/logging.properties` file. Any session running as `HR` and performing activation of `LogManager` will have the `LogManager` configured with properties coming from this database resident resource private to `HR`.

> **Note:**
>
> Oracle JVM always runs with a security manager. So, `HR` must be granted logging permissions, regardless of the logging configuration method used. In most cases, the following call issued by a privileged user is sufficient to grant these permissions:
>
> ```
> call dbms_java.grant_permission( 'HR',
> 'SYS:java.util.logging.LoggingPermission', 'control', '' );
> ```

2.5.4 Overview of Loading Classes Using the loadjava Tool

You can use the `loadjava` tool to create schema objects from files and load the schema objects to different schemas. For example,

```
loadjava -u HR -schema TEST MyClass.java
Password: password
```

> **Note:**
>
> You do *not* have to load the classes to the database as schema objects if you use the command-line interface. For example,
>
> ```
> C:\oraclehome\bin>loadjava -u HR MyClass.java
> Password: password
> ```

You can also run the `loadjava` tool from within SQL commands. Unlike a conventional JVM, which compiles and loads from files, Oracle JVM compiles and loads from database schema objects.

The following t able describes database schema objects that correspond to the files used by a conventional JVM.

Table 2-5 Description of Java Files

Java File Types	Description
`.java` source files	correspond to Java source schema objects
`.class` compiled Java files	correspond to Java class schema objects
`.properties` Java resource files or data files	correspond to Java resource schema objects

You must load all classes or resources into the database to be used by other classes within the database. In addition, at load time, you define who can run your classes within the database.

The following table describes the activities the `loadjava` tool performs for each type of file.

Table 2-6 loadjava Operations on Schema Objects

Schema Object	loadjava Operations on Objects
`.java` source files	1. Creates a Java source schema object in the definer's schema unless another schema is specified. 2. Loads the contents of the source file into a schema object. 3. Creates a class schema object for all classes defined in the source file. 4. If `-resolve` is requested, compiles the source schema object and resolves the class and its dependencies. It then stores the compiled class into a class schema object.
`.class` compiled Java files	1. Creates a class schema object in the definer's schema unless another schema is specified. 2. Loads the class file into the schema object. 3. Resolves and verifies the class and its dependencies if `-resolve` is specified.
`.properties` Java resource files	1. Creates a resource schema object in the definer's schema unless another schema is specified. 2. Loads a resource file into a schema object.

Note:

The `dropjava` tool performs the reverse of the `loadjava` tool. It deletes schema objects that correspond to Java files. Always use the `dropjava` tool to delete a Java schema object created with the `loadjava` tool. For example,

```
dropjava -u HR -schema TEST MyClass.java
Password: password
```

Dropping with SQL data definition language (DDL) commands will not update the auxiliary data maintained by the `loadjava` tool and the `dropjava` tool. You can also run the `dropjava` tool from within SQL commands.

After loading the classes and resources, you can access the `USER_OBJECTS` view in your database schema to verify whether your classes and resources have been loaded properly.

Related Topics

- Schema Objects and Oracle JVM Utilities

Related Topics

- About Using the Command-Line Interface

2.5.4.1 About Sharing of Metadata for User Classloaded Classes

Classes loaded by the built-in mechanism for loading database resident classes are known as **system classloaded**, whereas those loaded by other means are called **user classloaded**. When you load a class into the database, a representation of the class is created in memory, part of which is referred to here as the class metadata. The class metadata is the same for any session using the class and is potentially sharable. Earlier, such sharing was available only for system classloaded classes. Since Oracle Database 11*g*, you can also share class metadata of user classloaded classes, at the discretion of the system administrator.

Related Topics

- Classpath Extensions and User Classloaded Metadata

2.5.4.2 Defining the Same Class Twice

You cannot have two class objects with the same name in the same schema. This rule affects you in two ways:

> **Note:**
>
> Exceptions to this rule are:
>
> - When you use the `-prependjarnames` option for database resident JAR files. If you use this option, then you can have two classes with the same class name in the same schema.
> - When you load classes that are contained in modules. In such a case, different modules in the same database schema may contain definitions of the same class, but you may use only one definition of a class or package in any particular Oracle JVM session.

- You can load either a particular Java `.class` file or its `.java` file, but not both.

  Oracle Database tracks whether you loaded a class file or a source file. If you want to update the class, then you must load the same type of file that you originally loaded. If you want to update the other type, then you must drop the first before loading the second. For example, if you loaded `x.java` as the source for class `y`, then to load `x.class`, you must first drop `x.java`.
- You cannot define the same class within two different schema objects in the same schema. For example, suppose `x.java` defines class `y` and you want to move the definition of `y` to `z.java`. If `x.java` has already been loaded, then the `loadjava` tool rejects any attempt to load `z.java`, which also defines `y`. Instead, do either of the following:
  - Drop `x.java`, load `z.java`, which defines `y`, and then load the new `x.java`, which does not define `y`.
  - Load the new `x.java`, which does not define `y`, and then load `z.java`, which defines `y`.

Related Topics

- Database Resident JARs

2.5.4.3 About Designating Database Privileges and JVM Permissions

You must have the following SQL database privileges to load classes:

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.
- `oracle.aurora.security.JServerPermission.loadLibraryInClass.` *`classname`*.

Related Topics

- Permission for Loading Classes

2.5.4.4 About Loading JAR or ZIP Files

The `loadjava` tool accepts `.class`, `.java`, `.properties`, `.jar`, or `.zip` files. The JAR or ZIP files can contain source, class, and data files. When you pass a JAR or ZIP file to the `loadjava` tool, it opens the archive and loads the members of the archive individually. There is no JAR or ZIP schema object. If the JAR or ZIP content has not changed since the last time it was loaded, then it is not reloaded. Therefore, there is little performance penalty for loading JAR or ZIP files. In fact, loading JAR or ZIP files is the simplest way to use the `loadjava` tool.

> **Note:**
>
> Oracle Database does not reload a class if it has not changed since the last load. However, you can force a class to be reloaded using the `-force` option.
>
> Starting with Oracle Database Release 23ai, a JAR that is loaded by `loadjava`, can contain a module. Only a single module may be defined in the JAR, and the module definition must be at *top level* in it. In other words, `loadjava` enforces the same restrictions on the module JARs as client-side JAVA does when JARs are placed on its module path. If the `loadjava -automatic` option is specified, when a JAR with no `module-info` is loaded, then an *automatic* module is created from the entire JAR content. This is roughly equivalent to what occurs when a JAR that contains no `module-info` is placed on the client-side JAVA module path and loaded.
>
> Starting with the current release, loadjava supports *multi-release* JARs, including multi-release module JARs. A multi-release JAR contains alternate versions of classes that are chosen according to the current JDK release. In Oracle Database Release 23ai, appropriate versions of classes and resources for the greatest JDK release number that is less than or equal to 11, are loaded as database objects.

2.5.4.5 Database Resident JARs

When you load the contents of a JAR into the database, you have the option of creating a database object representing the JAR itself. In this way, you can retain an association between this JAR object and the class, resource, and source objects loaded from the JAR. This enables you to:

- Use signed JARs and JAR namespace segregation in the same way as you use them in standard JVM.
- Manage the classes that you have derived from a JAR while loading it into the database as a single unit. This helps you to prevent individual redefinition of the classes loaded from

the JAR. It also enables you to drop the whole set of classes loaded from the JAR, irrespective of the contents or the continued existence of the JAR on the external file system, at the time of dropping it.

In order to load a JAR into the database, you have the following options of the `loadjava` tool:

- `-jarsasdbobjects`
- `-prependjarnames`

When `loadjava` loads a JAR that is a multi-release JAR or a JAR that contains a module (or both), then the `-jarsasdbobjects` option is implied.

Related Topics

- The loadjava Tool

2.5.5 Overview of Granting Execute Rights

If you load all classes within your own schema and do not reference any class outside your schema, then you already have rights to run the classes. You have the privileges necessary for your objects to call other objects loaded in the same schema. That is, the ability for class `A` to call class `B`. Class `A` must be given the right to call class `B`.

The classes that define a Java application are stored within Oracle Database under the SQL schema of their owner. By default, classes that reside in one user's schema cannot be run by other users, because of security concerns. You can provide other users the right to run your class in the following ways:

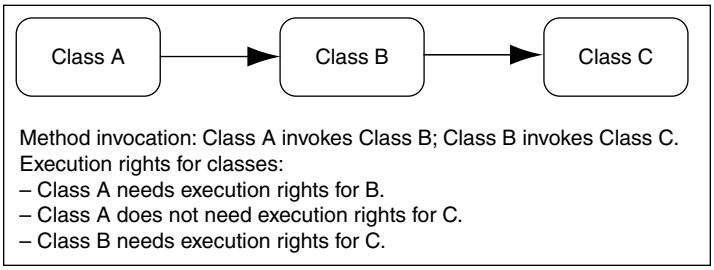
- Using the `loadjava -grant` option
- Using the following command:

```
SQL> grant execute on myclass to HR;
```

  where, `myclass` is the name of the underlying Java class.

The following figure illustrates the rights required to run classes.

Figure 2-3 Rights to Run Classes



Related Topics

- The loadjava Tool

Related Topics

- Oracle Database Java Application Performance

2.5.6 Overview of Controlling the Current User

During the execution of PL/SQL code, there is always a current user. The same concept is used for the execution of Java code. Initially, the current user is the user, who creates the session that invokes the Java code. A Java method is called from SQL or PL/SQL through a corresponding wrapper. Java wrappers are special PL/SQL entities, which expose Java methods to SQL and PL/SQL as PL/SQL stored procedures or functions. Such a wrapper might change the current effective user. The wrappers that change the current effective user to the owner of the wrapper are called definer's rights wrappers. If a wrapper does not change the current effective user, then the effective user remains unchanged.

By default, Java wrappers are definer's rights wrappers. If you want to override this, then create the wrapper using the `AUTHID CURRENT_USER` option.

At any time during the execution of Java code, a Java call stack is maintained. The stack contains frames corresponding to Java methods entered, with the innermost frame corresponding to the currently executing method. By default, Java methods execute on the stack without changing the current user, that is, with the privileges of their current effective invoker, not their definer.

You can load a Java class to the database with the `loadjava -definer` option. Any method of a class having the definer attribute marked, becomes a definer's rights method. When such a method is entered, a special kind of frame called a definer's frame is created onto the Java stack. This frame switches the current effective user to the owner (definer) of such a class. A new user ID remains effective for all inner frames until either the definer's frame is popped off the stack or a nested definer's frame is entered.

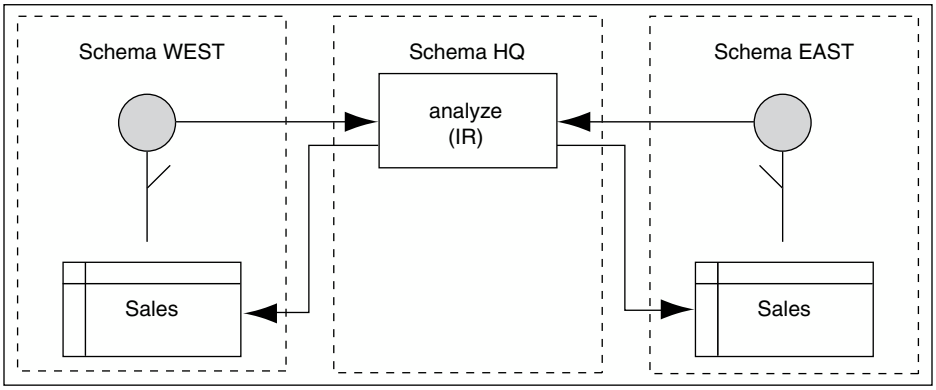
Thus, at any given time during the execution of a Java method that is called from SQL or PL/SQL through its wrapper, the effective user is one of the following:

- The innermost definer's frame on the Java stack
- Either the owner of the PL/SQL wrapper of the topmost Java method, if it is definer's rights, or the user who called the wrapper.

Consider a company that uses a definer's rights procedure to analyze sales. To provide local sales statistics, the procedure `analyze` must access `sales` tables that reside at each regional site. To do this, the procedure must also reside at each regional site. This causes a maintenance problem. To solve the problem, the company installs an invoker's rights version of the procedure `analyze` at headquarters.

The following figure shows how all regional sites can use the same procedure to query their own `sales` tables.

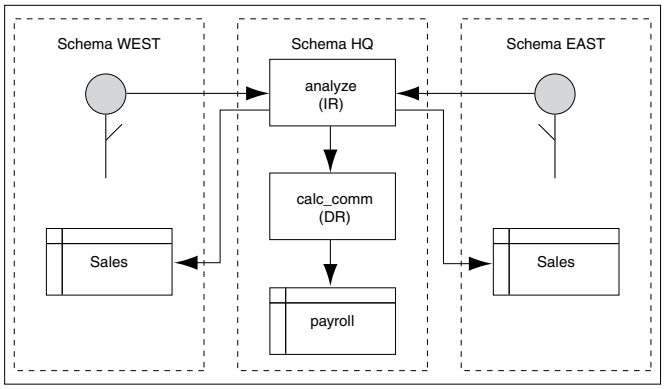
Figure 2-4 Invoker's rights Solution



Occasionally, you may want to override the default invoker's rights behavior. Suppose headquarters wants the `analyze` procedure to calculate sales commissions and update a central `payroll` table. This presents a problem, because invokers of `analyze` should not have direct access to the `payroll` table, which stores employee salaries and other sensitive data.

The following figure illustrates the solution, where the `analyze` procedure call the definer's rights procedure, `calcComm`, which in turn updates the `payroll` table.

Figure 2-5 Indirect Access



Related Topics

- Writing Top-Level Call Specifications
  This section describes how to define top-level call specifications in SQL*Plus.

2.5.7 Overview of Checking Java Uploads

You can query the `USER_OBJECTS` database view to obtain information about schema objects that you own, including Java sources, classes, and resources. This enables you, for example, to verify whether sources, classes, or resources that you load are properly stored in schema objects.

The following table lists the key columns in `USER_OBJECTS` and their description.

Table 2-7 Key USER_OBJECT Columns

Name	Description
OBJECT_NAME	Name of the object
OBJECT_TYPE	Type of the object, such as JAVA SOURCE, JAVA CLASS, or JAVA RESOURCE.
STATUS	Status of the object. The values can be either VALID or INVALID. It is always VALID for JAVA RESOURCE.

Object Name and Type

An OBJECT_NAME in USER_OBJECTS is the alias. The fully qualified name is stored as an alias if it exceeds 30 characters.

If the server uses an alias for a schema object, then you can use the LONGNAME() function of the DBMS_JAVA package to receive it from a query as a fully qualified name, without having to know the alias or the conversion rules.

```
SQL> SELECT dbms_java.longname(object_name) FROM user_objects WHERE object_type='JAVA
SOURCE';
```

This statement displays the fully qualified name of the Java source schema objects. Where no alias is used, no conversion occurs.

> **Note:**
>
> SQL and PL/SQL are *not* case-sensitive.

You can use the SHORTNAME() function of the DBMS_JAVA package to use a fully qualified name as a query criterion, without having to know whether it was converted to an alias in the database.

```
SQL*Plus> SELECT object_type FROM user_objects WHERE
object_name=dbms_java.shortname('known_fullname');
```

This statement displays the OBJECT_TYPE of the schema object with the specified fully qualified name. This presumes that the fully qualified name is representable in the database character set.

```
SQL> select * from javasnm;
SHORT LONGNAME
--------------------------------------------------------------------------
/78e6d350_BinaryExceptionHandl sun/tools/java/BinaryExceptionHandler
/b6c774bb_ClassDeclaration sun/tools/java/ClassDeclaration
/af5a8ef3_JarVerifierStream1 sun/tools/jar/JarVerifierStream$1
```

This statement displays all the data stored in the javasnm view.

Status

STATUS is a character string that indicates the validity of a Java schema object. A Java source schema object is VALID if it compiled successfully, and a Java class schema object is VALID if it

was resolved successfully. A Java resource schema object is always `VALID`, because resources are not resolved.

Example: Accessing USER_OBJECTS

The following SQL*Plus script accesses the `USER_OBJECTS` view to display information about uploaded Java sources, classes, and resources:

```
COL object_name format a30
COL object_type format a15
SELECT object_name, object_type, status
       FROM user_objects
       WHERE object_type IN ('JAVA SOURCE', 'JAVA CLASS', 'JAVA RESOURCE')
       ORDER BY object_type, object_name;
```

You can optionally use wildcards in querying `USER_OBJECTS`, as in the following example:

```
SELECT object_name, object_type, status
       FROM user_objects
       WHERE object_name LIKE '%Alerter';
```

The preceding statement finds any `OBJECT_NAME` entries that end with the characters `Alerter`.

Related Topics

- Shortened Class Names
  Each Java source, class, and resource is stored in its own schema object in the server. The name of the schema object is derived from the fully qualified name, which includes relevant path or package information. Dots are replaced by slashes.

2.5.8 About Publishing Java Methods Loaded in the Database

Oracle Database enables clients and SQL to call Java methods that are loaded in the database after they are published. You publish either the object itself or individual methods. If you write a Java stored procedure that you intend to call with a trigger, directly or indirectly in SQL data manipulation language (DML) or in PL/SQL, then you must publish individual methods in the class. Using a call specification, specify how to access the method. Java programs consist of many methods in many classes. However, only a few `static` methods are typically exposed with call specifications.

Related Topics

- Publishing Java Classes With Call Specifications

2.5.9 Overview of Auditing Java Classes Loaded in the Database

You can audit DDL statements for creating, altering, or dropping Java source, class, and resource schema objects, as with any other DDL statement. Oracle Database provides auditing options for auditing Java activities easily and directly. You can also audit any modification of Java sources, classes, and resources.

You can audit database activities related to Java schema objects at two different levels, statement level and object level. At the statement level you can audit all activities related to a special pattern of statements.

Table 2-8 lists the statement auditing options and the corresponding SQL statements related to Java schema objects.

Table 2-8 Statement Auditing Options Related to Java Schema Objects

Statement Option	SQL Statements
CREATE JAVA SOURCE	CREATE JAVA SOURCE CREATE OR REPLACE JAVA SOURCE
ALTER JAVA SOURCE	ALTER JAVA SOURCE
DROP JAVA SOURCE	DROP JAVA SOURCE
CREATE JAVA CLASS	CREATE JAVA CLASS CREATE OR REPLACE JAVA CLASS
ALTER JAVA CLASS	ALTER JAVA CLASS
DROP JAVA CLASS	DROP JAVA CLASS
CREATE JAVA RESOURCE	CREATE JAVA RESOURCE CREATE OR REPLACE JAVA RESOURCE
ALTER JAVA RESOURCE	ALTER JAVA RESOURCE
DROP JAVA RESOURCE	DROP JAVA RESOURCE

For example, if you want to audit the `ALTER JAVA SOURCE` DDL statement, then enter the following statement at the SQL prompt:

```
AUDIT ALTER JAVA SOURCE
```

Object level auditing provides finer granularity. It enables you to identify specific problems by zooming into specific objects.

Table 2-9 lists the object auditing options for each Java schema object. The entry X in a cell indicates that the corresponding SQL command can be audited for that Java schema object. The entry NA indicates that the corresponding SQL command is not applicable for that Java schema object.

Table 2-9 Object Auditing Options Related to Java Schema Options

Object Option	Java Source	Java Resource	Java Class
ALTER	X	NA	X
EXECUTE	NA	NA	X
AUDIT	X	X	X
GRANT	X	X	X

See Also:

- *Oracle Database Security Guide*
- *Oracle Database SQL Language Reference*

2.6 User Interfaces on the Server

Oracle Database furnishes all core Java class libraries on the server, including those associated with presentation of the user interfaces. However, it is inappropriate for code running on the server to attempt to materialize or display a user interface on the server. Users running applications in Oracle JVM environment should not be expected nor allowed to interact with or depend on the display and input hardware of the server where Oracle Database is running.

To address compatibility issues on platforms that do not support display, keyboard, or mouse, Java 1.4 outlines Headless Abstract Window Toolkit (AWT) support. The Headless AWT API introduces a new `public` run-time exception class, `java.awt.HeadlessException`. The constructors of the `Applet` class, all heavy-weight components, and many of the methods in the `Toolkit` and `GraphicsEnvironment` classes, which rely on the native display devices, are changed to throw `HeadlessException` if the platform does not support a display. In Oracle Database, user interfaces are supported only on client applications. Accordingly, Oracle JVM is a Headless Platform and throws `HeadlessException` if these methods are called.

Most AWT computation that does not involve accessing the underlying native display or input devices is allowed in Headless AWT. In fact, Headless AWT is quite powerful as it provides programmers access to fonts, imaging, printing, and color and ICC manipulation. For example, applications running in Oracle JVM can parse, manipulate, and write out images as long as they do not try to physically display it on the server. The standard JVM implementation can be started in the Headless mode, by supplying the `-Djava.awt.headless=true` property, and run with the same Headless AWT restrictions as Oracle JVM does. Oracle JVM fully complies with the Java Compatibility Kit (JCK) with respect to Headless AWT.

> **See Also:**
>
> `http://www.oracle.com/technetwork/articles/javase/headless-136834.html`

Oracle JVM takes a similar approach for sound support. Applications in Oracle JVM are not allowed to access the underlying sound system for purposes of sound playback or recording. Instead, the system sound resources appear to be unavailable in a manner consistent with the sound API specification of the methods that are trying to access the resources. For example, methods in `javax.sound.midi.MidiSystem` that attempt to access the underlying system sound resources throw the `MidiUnavailableException` checked exception to signal that the system is unavailable. However, similar to the Headless AWT support, Oracle Database supports the APIs that allow sound file manipulation, free of the native sound devices. Oracle JVM also fully complies with the JCK, when it implements the sound API.

2.7 Shortened Class Names

Each Java source, class, and resource is stored in its own schema object in the server. The name of the schema object is derived from the fully qualified name, which includes relevant path or package information. Dots are replaced by slashes.

Schema object names, however, have a maximum of only 128 characters, and all characters must be legal and convertible to characters in the database character set. If any fully qualified name is longer than 128 characters or contains illegal or nonconvertible characters, then Oracle Database converts it to a short name, or alias, to use as the name of the schema

object. Oracle Database keeps track of both the names and how to convert between them. If the fully qualified name is 128 characters or less and has no illegal or inconvertible characters, then it is used as the schema object name.

Because Java classes and methods can have names exceeding the maximum SQL identifier length, Oracle Database uses abbreviated names internally for SQL access. Oracle Database provides the `LONGNAME()` function within the `DBMS_JAVA` package for retrieving the original Java class name for any truncated name.

```
FUNCTION longname (shortname VARCHAR2) RETURN VARCHAR2
```

This function returns the fully qualified name of the Java schema object, which is specified using its alias. The following is an example of a statement used to display the fully qualified name of classes that are invalid:

```
SELECT dbms_java.longname (object_name) FROM user_objects WHERE object_type = 'JAVA
CLASS' and status = 'INVALID';
```

You can also specify a full name to the database by using the `SHORTNAME()` function of the `DBMS_JAVA` package. The function takes a full name as input and returns the corresponding short name. This function is useful for verifying whether the classes are loaded successfully, by querying the `USER_OBJECTS` view.

```
FUNCTION shortname (longname VARCHAR2) RETURN VARCHAR2
```

Related Topics

- System Classes

2.8 Class.forName() in Oracle Database

The JLS provides the following description of `Class.forName()`:

Given the fully qualified name of a class, this method attempts to locate, load, and link the class. If it succeeds, then a reference to the `Class` object for the class is returned. If it fails, then an instance of `ClassNotFoundException` is thrown.

Class lookup is always on behalf of a referencing class and is done through an instance of `ClassLoader`. The difference between the Java Development Kit (JDK) implementation and Oracle JVM implementation is the method in which the class is found:

- The JDK uses one instance of `ClassLoader` that searches the set of directory tree roots specified by the `CLASSPATH` environment variable.
- Oracle JVM defines several resolvers that specify how to locate classes. Every class has a resolver associated with it, and each class can, potentially, have a different resolver. When you run a method that calls `Class.forName()`, the resolver of the currently running class, which is `this`, is used to locate the class.

You can receive unexpected results if you try to locate a class with an incorrect resolver. For example, if a class `X` in schema `X` requests a class `Y` in schema `Y` to look up class `Z`, you will experience an error if you expected the resolver of class `X` to be used. Because class `Y` is performing the lookup, the resolver associated with class `Y` is used to locate class `Z`. In summary, if the class exists in another schema and you specified different resolvers for different classes, as would happen by default if they are in different schemas, you may not find the class.

You can solve this resolver problem as follows:

- Avoid any class name lookup by passing the `Class` object itself.

- Supply the `ClassLoader` instance in the `Class.forName()` method.
- Supply the class and the schema it resides in to the `classForNameAndSchema()` method.
- Supply the schema and class name to `ClassForName.lookupClass()`.
- Serialize your objects with the schema name and the class name.

> **Note:**
>
> Another unexpected behavior can occur if system classes invoke `Class.forName()`. The desired class is found only if it resides in `SYS` or in `PUBLIC`. If your class does not exist in either `SYS` or `PUBLIC`, then you can declare a `PUBLIC` synonym for the class.

This section covers the following topics:

- Supply ClassLoader in Class.forName()
- Supply Class and Schema Names to classForNameAndSchema()
- Supply Class and Schema Names to lookupClass()
- Supply Class and Schema Names when Serializing
- Class.forName Example

Related Topics

- Overview of Resolving Class Dependencies
  Many Java classes contain references to other classes, which is the essence of reusing code. A conventional JVM searches for `.class`, `.zip`, and `.jar` files within the directories specified in `CLASSPATH`.

2.8.1 Supply ClassLoader in Class.forName()

Oracle Database uses resolvers for locating classes within schemas. Every class has a specified resolver associated with it, and each class can have a different resolver associated with it. As a result, the locating of classes is dependent on the definition of the associated resolver. The `ClassLoader` instance knows which resolver to use, based on the class that is specified. When you supply a `ClassLoader` instance to `Class.forName()`, your class is looked up in the schemas defined in the resolver of the class. The syntax of this variant of `Class.forName()` is as follows:

```
Class.forName (String name, boolean initialize, ClassLoader loader);
```

The following examples show how to supply the class loader of either the current class instance or the calling class instance.

Example 2-1 Retrieve Resolver from Current Class

You can retrieve the class loader of any instance by using the `Class.getClassLoader()` method. The following example retrieves the class loader of the class represented by instance `x`:

```
Class c1 = Class.forName (x.whatClass(), true, x.getClass().getClassLoader());
```

Example 2-2 Retrieve Resolver from Calling Class

You can retrieve the class of the instance that called the running method by using the `oracle.aurora.vm.OracleRuntime.getCallerClass()` method. After you retrieve the class, call the `Class.getClassLoader()` method on the returned class. The following example retrieves the class of the instance that called the `workForCaller()` method. Then, its class loader is retrieved and supplied to the `Class.forName()` method. As a result, the resolver used for looking up the class is the resolver of the calling class.

```
void workForCaller()
{
  ClassLoader c1=oracle.aurora.vm.OracleRuntime.getCallerClass().getClassLoader();
  ...
  Class c=Class.forName(name, true, c1);
  ...
}
```

2.8.2 Supply Class and Schema Names to classForNameAndSchema()

You can resolve the problem of where to find the class by supplying the resolver, which can identify the schemas to be searched. Alternatively, you can supply the schema in which the class is loaded. If you know in which schema the class is loaded, then you can use the `classForNameAndSchema()` method, which is in the `DbmsJava` class provided by Oracle Database. This method takes both the name of the class and the schema in which the class resides and locates the class within the designated schema.

Example 2-3 Providing Schema and Class Names

The following example shows how you can save the schema and class names using the `save()` method. Both names are retrieved, and the class is located using the `DbmsJava.classForNameAndSchema()` method.

```
import oracle.aurora.rdbms.ClassHandle;
import oracle.aurora.rdbms.Schema;
import oracle.aurora.rdbms.DbmsJava;

void save (Class c1)
{
  ClassHandle handle = ClassHandle.lookup(c1);
  Schema schema = handle.schema();
  writeName (schema.getName());
  writeName (c1.getName());
}

Class restore()
{
  String schemaName = readName();
  String className = readName();
  return DbmsJava.classForNameAndSchema (schemaName, className);
}
```

2.8.3 Supply Class and Schema Names to lookupClass()

You can supply a `String` value containing both the schema and class names to the `oracle.aurora.util.ClassForName.lookupClass()` method. When called, this method locates the class in the specified schema. The string must be in the following format:

```
"<schema>:<class>"
```

For example, to locate `com.package.myclass` in the `HR` schema, use the following:

```
oracle.aurora.util.ClassForName.lookupClass("HR:com.package.myclass");
```

> **Note:**
>
> Use uppercase characters for the schema name. In this case, the schema name is case-sensitive.

2.8.4 Supply Class and Schema Names when Serializing

When you deserialize a class, part of the operation is to lookup a class based on a name. To ensure that the lookup is successful, the serialized object must contain both the class and schema names.

Oracle Database provides the following classes for serializing and deserializing objects:

- `oracle.aurora.rdbms.DbmsObjectOutputStream`

  This class extends `java.io.ObjectOutputStream` and adds schema names in the appropriate places.

- `oracle.aurora.rdbms.DbmsObjectInputStream`

  This class extends `java.io.ObjectInputStream` and reads streams written by `DbmsObjectOutputStream`. You can use this class in any environment. If used within Oracle Database, then the schema names are read out and used when performing the class lookup. If used on a client, then the schema names are ignored.

2.8.5 Class.forName Example

The following example shows several methods for looking up a class:

```
import oracle.aurora.vm.OracleRuntime;
import oracle.aurora.rdbms.Schema;
import oracle.aurora.rdbms.DbmsJava;

public class ForName
{
  private Class from;

  /* Supply an explicit class to the constructor */
  public ForName(Class from)
  {
    this.from = from;
  }

  /* Use the class of the code containing the "new ForName()" */
  public ForName()
  {
    from = OracleRuntime.getCallerClass();
  }

  /* lookup relative to Class supplied to constructor */
  public Class lookupWithClassLoader(String name) throws ClassNotFoundException
  {
    /* A ClassLoader uses the resolver associated with the class*/
    return Class.forName(name, true, from.getClassLoader());
```

```
  }

  /* In case the schema containing the class is known */
  static Class lookupWithSchema(String name, String schema)
  {
    Schema s = Schema.lookup(schema);
    return DbmsJava.classForNameAndSchema(name, s);
  }
}
```

The preceding example uses the following methods for locating a class:

- To use the resolver of the class of an instance, call `lookupWithClassLoader()`. This method supplies a class loader to the `Class.forName()` method in the `from` variable. The class loader specified in the `from` variable defaults to this class.
- To use the resolver from a specific class, call `ForName()` with the designated class name, followed by `lookupWithClassLoader()`. The `ForName()` method sets the `from` variable to the specified class. The `lookupWithClassLoader()` method uses the class loader from the specified class.
- To use the resolver from the calling class, first call the `ForName()` method without any parameters. It sets the `from` variable to the calling class. Then, call the `lookupWithClassLoader()` method to locate the class using the resolver of the calling class.
- To lookup a class in a specified schema, call the `lookupWithSchema()` method. This provides the class and schema name to the `classForNameAndSchema()` method.

2.9 About Managing Your Operating System Resources

Operating system resources are a limited commodity on any computer. Because Java is targeted at providing a computing platform as well as a programming language, it contains platform-independent classes and frameworks for accessing platform-specific resources. The Java class methods access operating system resources through JVM. Java has potential problems with this model because programmers rely on the garbage collector to manage all resources, when all that the garbage collector manages is Java objects and not the operating system resources that the Java objects hold on to.

In addition, when you use shared servers, your operating system resources, which are contained within Java objects, can be invalidated if they are maintained across calls within a session.

The following sections discuss these potential problems:

- Overview of Operating System Resources
- Garbage Collection and Operating System Resources

Related Topics

- Operating System Resources Affected Across Calls
  In the shared server mode, Oracle JVM closes any open operating system resources at the end of a database call.

2.9.1 Overview of Operating System Resources

In general, your operating system resources contain the following:

Operating System Resources	Description
memory	Oracle Database manages memory internally, allocating memory as you create objects and freeing objects as you no longer need them. The language and class libraries do not support a direct means to allocate and free memory.
files and sockets	Java contains classes that represent file or socket resources. Instances of these classes hold on to the file or socket constructs, such as file handles, of the operating system.
threads	Oracle JVM threads provide no additional scalability over what is provided by the database support of multiple concurrently executing sessions. However, Oracle JVM supports the full Java threading API.

Operating System Resource Access

By default, a Java user does not have direct access to most operating system resources. A system administrator can give permissions to a user to access these resources by modifying JVM security restrictions. JVM security enforced upon system resources conforms to Java 2 security.

Operating System Resource Lifetime

You can access operating system resources using the standard core Java classes and methods. Once you access a resource, the time that it remains active varies according to the type of resource. Memory is garbage collected. Files, threads, and sockets persist across calls when you use a dedicated mode server. In shared server mode, files, threads, and sockets terminate when the call ends.

Related Topics

- Overview of Java Security Features
- Operating System Resources Affected Across Calls
  In the shared server mode, Oracle JVM closes any open operating system resources at the end of a database call.

2.9.2 Garbage Collection and Operating System Resources

Imagine that memory is divided into two realms: Java object memory and operating system constructs. The Java object memory realm contains all objects and variables. Operating system constructs include resources that the operating system allocates to the object when it asks. These resources include files, sockets, and so on.

Basic programming rules dictate that you close all memory, both Java objects and operating system constructs. Java programmers incorrectly assume that memory is freed by the garbage collector. The garbage collector was created to collect all unused Java object memory. However, it does not close operating system constructs. All operating system constructs must be closed by the program before the Java object is garbage collected.

For example, whenever an object opens a file, the operating system creates the file and gives the object a file handle. If the file is not closed, then the operating system holds the file handle construct open until the call ends or JVM exits. This may cause you to run out of these constructs earlier than necessary. There are a finite number of handles within each operating system. To guarantee that you do not run out of handles, close your resources before exiting the method. This includes closing the streams attached to your sockets before closing the socket.

For performance reasons, the garbage collector cannot examine each object to see if it contains a handle. As a result, the garbage collector collects Java objects and variables, but does not issue the appropriate operating system methods for freeing any handles.

Example 2-4 shows how to close the operating system constructs.

If you do not close `inFile`, then eventually the `File` object will be garbage collected. Even after the `File` object is garbage collected, the operating system treats the file as if it were in use, because it was not closed.

> **Note:**
>
> You may want to use Java finalizers to close resources. However, finalizers are not guaranteed to run in a timely manner. Instead, finalizers are put on a queue to run when the garbage collector has time. If you close your resources within your finalizer, then it might not be freed until JVM exits. The best approach is to close your resources within the method.

Example 2-4 Closing Your Operating System Resources

```
public static void addFile(String[] newFile)
{
  File inFile = new File(newFile);
  FileReader in = new FileReader(inFile);
  int i;

  while ((i = in.read()) != -1)
    out.write(i);

  /*closing the file, which frees up the operating system file handle*/
  in.close();
}
```

2.10 About Using the Runtime.exec Functionality in Oracle Database

Java Virtual Machine fully supports the family of Java Standard Edition `java.lang.Runtime.exec` methods. These methods spawn a new operating system (OS) process to run a user-supplied command. On the server, you must use these methods with caution. In Java Virtual Machine, OS command execution permissions are not granted to all database users by default and are issued only by privileged administrators. If you are a DBA, then you must know how to use the `Runtime.exec` functionality in Oracle Database and follow the recommendations. Also, you must be selective about issuing these permissions to database users.

Related Topics

- Secure Use of Runtime.exec Functionality in Oracle Database

2.11 Managing Your Applications Using JMX

This section contain the following topics:

- Overview of JMX

2.11.1 Overview of JMX

JMX (Java Management Extensions) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices, service-oriented networks, and JVM (Java Virtual Machine). This API allows its classes to be dynamically constructed and changed. So, you can use this technology to monitor and manage resources as they are created, installed, and implemented. The JMX API also includes remote access, so a remote management program can interact with a running application for these purposes.

In JMX, a given resource is instrumented by one or more Java objects known as MBeans (Managed Beans). These MBeans are registered in a core managed object server, known as an MBean server, that acts as a management agent and can run on most devices enabled for the Java programming language. A JMX agent consists of an MBean server, in which MBeans are registered, and a set of services for handling MBeans.

See Also:

- http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html
- http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html

2.11.2 Enabling and Starting JMX in a Session

To help in enabling and running JMX services in sessions running Java, the `JMXSERVER` role and the `dbms_java.start_jmx_agent` procedure are provided. The `JMXSERVER` role is granted specific Java permissions that enable you to start and run MBeanServer and JMX agent in a session. The procedure `dbms_java.start_jmx_agent` starts the agent in a specific session that generally remains active for the duration of the session. Perform the following to enable and start JMX:

1. Obtain `JMXSERVER` from `SYS` or `SYSTEM`:

   ```
   SQL> grant jmxserver to HR;
   ```

   where, `HR` is the user name.

2. Invoke the procedure `dbms_java.start_jmx_agent` to startup JMX in the session. The `dbms_java.start_jmx_agent` procedure can be invoked with the following arguments:

   `port`: the port for the JMX listener. The value of this parameter sets the Java property `com.sun.management.jmxremote.port`.

`ssl`: sets the value for the Java property `com.sun.management.jmxremote.ssl`. Case for `true` and `false` values is ignored.

`auth`: the value for the property `com.sun.management.jmxremote.authenticate`, otherwise a semicolon-separated list of Java Authentication and Authorization Service (JAAS) credentials. The value is not case-sensitive.

Each of these arguments can be `null` or omitted, with `null` as the default value. when an argument is `null`, it does not alter the previously present value of the corresponding property in the session.

> **Note:**
>
> The Java properties corresponding to the parameters of `dbms_java.start_jmx_agent` are from the set of Java properties specified in standard Java 5.0 JMX documentation. For the full list of Java JMX properties please refer to
>
> `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`

The `dbms_java.start_jmx_agent` procedure starts an agent activating OJVM JMX server and a listener. The JMX server runs as one or more daemon threads in the current session and in general is available for the duration of the session. Once JMX Agent is started in a session, Java code running in the session can be monitored.

The `dbms_java.start_jmx_agent` procedure is a PL/SQL wrapper for the Java method `oracle.aurora.rdbms.JMXAgent.startOJVMAgent`, which by itself can also be called programmatically from Java stored procedures. The `startOJVMAgent` method starts the JMX Server and the JMX connectivity daemon threads, and then exits. On dedicated servers, these threads may remain active for the duration of the session, but go into an inert state for the time intervals between calls. When these intervals are short, then the same socket connections resume transparently. This enables clients such as JConsole to remain connected across multiple calls.

A different mode of JMX monitoring is possible with the `EXIT_CALL_WHEN_ALL_THREADS_TERMINATE` policy. By setting the call exit policy to `OracleRuntime.EXIT_CALL_WHEN_ALL_THREADS_TERMINATE`, you can configure the session to run JMX server continuously in a call that invokes the `startOJVMAgent` method till the Java call is exited programmatically. This mode is convenient when various Java tasks are fired up from a JMX client as operations of specific MBeans. This way, continuous JMX management and monitoring is driven by these operations. Please refer to the JVM JMX demo for such a bean, for example, `jmxserv.Load`.

Related Topics

- Shared Server Limitations for JMX

2.11.3 Setting Oracle JVM JMX Defaults and Configurability

When `dbms_java.start_jmx_agent` is activated, the `com.sun.management.jmxremote` property is set to `true`.

Before invoking `start_jmx_agent`, a JMXSERVER-privileged user can preset various management properties in the following ways:

- Using the `dbms_java.set_property` PL/SQL function
- Invoking the `java.lang.System.setProperty` method

The `JMXSERVER` role user can also preset the properties in the Database resident Java resource, using the `com.sun.management.config.file` Java property. If you do not set this property, then the default name of the resource is `lib/management/management.properties`. This default resource, provided in the SYS schema, is suitable for most use scenarios. An alternative name, specified by the `com.sun.management.config.file` Java property, is first looked up in the user's own schema, and then in the SYS schema.

This resource mechanism is the Oracle JVM extension of standard file-based JMX configuration management. This mechanism works better for Oracle JVM as it provides more security and per-schema management. When the resource specified with the `com.sun.management.config.file` does not exist in the user's own schema or in the SYS schema, then a file-read is attempted as a fallback, and the name is prefixed with `$(java.home)/`. In this release, the default values provded in the `lib/management/management.properties` file are:

```
com.sun.management.jmxremote.ssl.need.client.auth = true
com.sun.management.jmxremote.authenticate = false
```

The property `com.sun.management.jmxremote.ssl.need.client.auth` in conjunction with `com.sun.management.jmxremote.ssl`, sets JMX for two-way encrypted SSL authentication with client and server certificates. The default value of `com.sun.management.jmxremote.ssl` is `true`. This configuration is the default and is preferred over JAAS password authentication.

> **Note:**
>
> For more information visit the following:
>
> - http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html
> - http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html

> **Note:**
>
> The default JMX Login Module providing file-based store for passwords is not supported in Oracle JVM for security reasons. So, if JAAS password authentication must be used instead of SSL client authentication, then pass transient JAAS credentials securely as the `auth` parameter to `dbms_java.start_jmx_agent` as illustrated in this section, or configure JMX to use a secure custom LDAP login module.

2.11.4 Examples of SQL calls to dbms_java.start_jmx_agent

Following are some examples of starting the JMX server:

- Starts the JMX server and the listener using the default settings as described in the preceding sections or the values set earlier in the same session:

```
call dbms_java.start_jmx_agent();
```

- Starts the JMX server and the listener using the default settings as described in the preceding sections or the values set earlier in the same session:

```
call dbms_java.start_jmx_agent(null, null, null);
```

- Starts the JMX server and the listener on port 9999 with the other JMX settings having the default values or the values set earlier in the same session:

```
call dbms_java.start_jmx_agent('9999');
```

- Starts the JMX server and the listener on port 9999 with the other JMX settings having the default values or the values set earlier in the same session:

```
call dbms_java.start_jmx_agent('9999', null, null);
```

- Starts the JMX server and the listener with the JMX settings having the default values or the values set earlier in the same session and with JAAS credentials `monitorRole/1z2x` and `controlRole/2p3o`:

```
call dbms_java.start_jmx_agent(null, null, 'monitorRole/1z2x;controlRole/2p3o');
```

  These credentials are transient. The property `com.sun.management.jmxremote.authenticate` is set to `true`.

- Starts JMX listener on port 9999 with no SSL and no JAAS authentication. Used only for development or demonstration.

```
call dbms_java.start_jmx_agent('9999', 'false', 'false');
```

Related Topics

- Important Security Notes

2.11.5 Using JConsole to Monitor and Control Oracle JVM

This section describes how to use JConsole, a standard JMX client tool, for monitoring and controlling Oracle JVM. JConsole is a part of standard Java JDK.

This section discusses the following topics:

- Using the jconsole Command
- About Using the JConsole interface
- The OracleRuntime MBean
- Memory Thresholds

> **Note:**
>
> To monitor Java in the database with JConsole, you should have a server-side Java session running JMX Agent.

Related Topics

- Enabling and Starting JMX in a Session

2.11.5.1 Using the jconsole Command

Use the `jconsole` command syntax to start JConsole. The simplest format to start the JConsole tool is the following:

```
jconsole [hostName:portNum]
```

where:

- `hostname` is the name of the system running the application
- `portNum` is the port number of the JMX listener

In the following examples, we connect to a host with name `example.com` through default port 9999. This mode assumes no authentication and encryption. This mode is adequate only for demo or testing, and can be used to connect to Oracle JVM JMX sessions that are started with the following command:

```
call dbms_java.start_jmx_agent(portNum, false, false);
```

Remember that you can connect to and interact with Oracle JVM from JConsole, only when the daemon threads of the server are running and are not dormant. This means that there should be an active Java call in the session, which is running the JMX server on the specified port. During the time interval between subsequent Java calls, JMX server preserves its state and statistics, but is unable to communicate with JConsole.

Related Topics

- Important Security Notes

2.11.5.2 About Using the JConsole interface

The JConsole interface consists of the following tabs:

- Summary tab

  It displays summary information on Oracle JVM and the values monitored by JMX.
- Memory tab

  It displays information on memory usage.
- Threads tab

  It displays information on thread usage.
- Classes tab

  It displays information on class loading.
- MBeans tab

  It displays information on MBeans.
- VM tab

  It displays information on Oracle JVM.

> **Note:**
>
> In the current release of Oracle Database, the data collected and passed to JConsole is limited to the Oracle JVM session that runs the JMX agent. This data does not include the attributes of other sessions that may be running in Oracle JVM. One exception is the `OracleRuntime MBean` that provides information about many `WholeJVM_ Attributes` and operations of Oracle JVM.

Related Topics

- The OracleRuntime MBean
  When the `dbms_java.start_jmx_agent` procedure is called, then `OracleRuntime` is added to the list of Oracle JVM platform MBeans. This MBean is specific to Oracle JVM.

2.11.5.3 About Viewing Oracle JVM Summary Information

You can use the Summary tab of the JConsole interface to view Oracle JVM Summary Information. This tab displays key monitoring information on thread usage, memory consumption, class loading, and other VM and operating system specifics.

If JConsole successfully connects to an Oracle JVM session running a JMX Agent, then the Overview Tab looks the following image:

Figure 2-6 The Overview Tab of the JConsole Interface

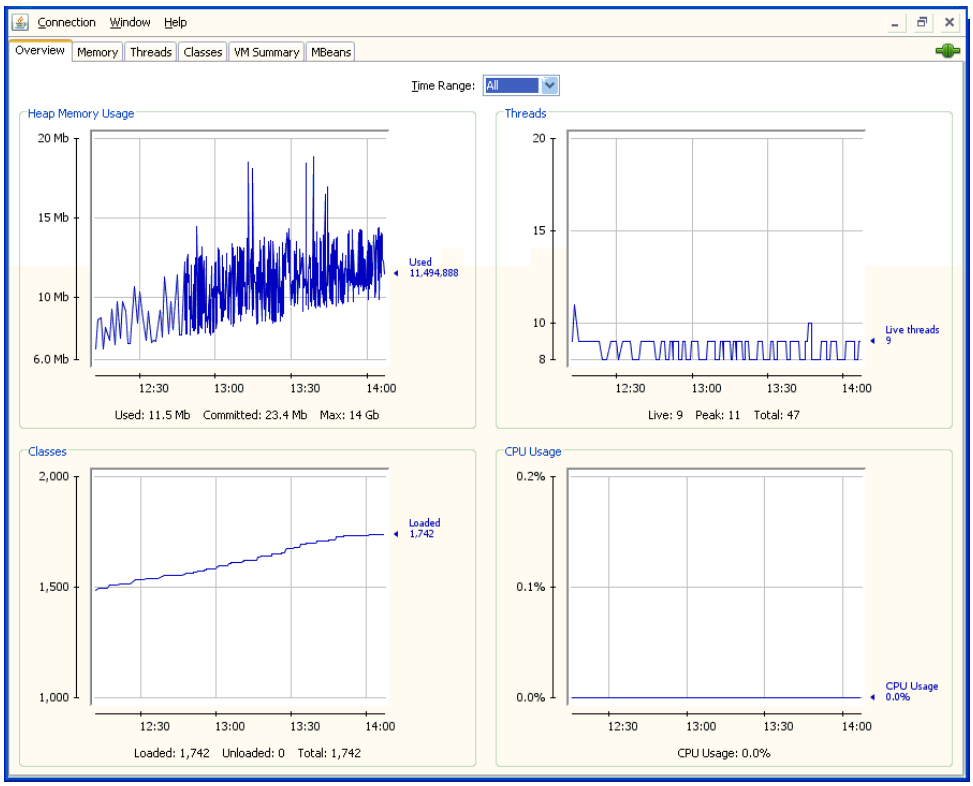


Table 2-10 provides description of the fields present in the Overview tab.

Table 2-10 Description of the Overview Tab Fields in JConsole Interface

Field	Description
Up time	The duration for which the Oracle JVM session has been running.
Process CPU time	This information is not gathered for Oracle JVM sessions in the current release of Oracle Database.
Live threads	The current number of live daemon and non-daemon threads.
Peak	Highest number of live threads since Oracle JVM started.

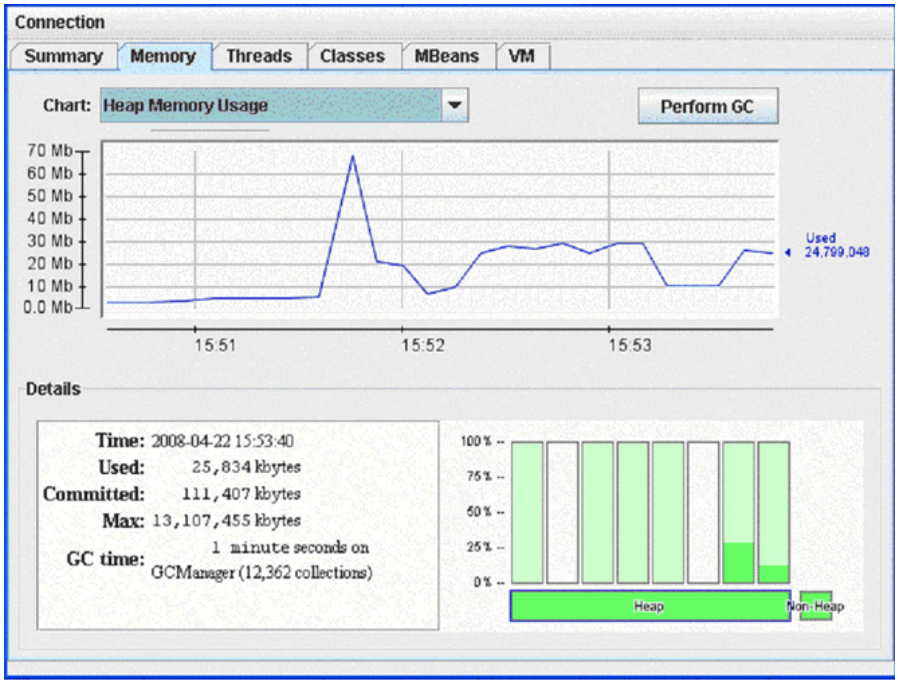
Table 2-10 (Cont.) Description of the Overview Tab Fields in JConsole Interface

Field	Description
Daemon threads	Current number of live daemon threads.
Total started	Total number of threads started since Oracle JVM started. It includes daemon, non-daemon, and terminated threads.
Current heap size	Number of kilobytes currently occupied by the heap.
Committed memory	Total amount of memory allocated for use by the heap.
Maximum heap size	Maximum number of kilobytes occupied by the heap.
Objects pending for finalization	Number of objects pending for finalization.
Garbage collector information	Information about the garbage collector, which includes name, number of collections performed, and total time spent performing garbage collection.
Current classes loaded	Number of classes currently loaded into memory for execution.
Total classes loaded	Total number of classes loaded into session memory since the session started.
Total classes unloaded	Number of classes unloaded from memory. Typically this is zero for the current release of Oracle Database.
Total physical memory	This information is not gathered for Oracle JVM sessions in the current release of Oracle Database. So, the value displayed is zero.
Free physical memory	This information is not gathered for Oracle JVM sessions in the current release of Oracle Database. So, the value displayed is zero.
Committed virtual memory	This information is not gathered for Oracle JVM sessions in the current release of Oracle Database. So, the value displayed is zero.

2.11.5.4 About Monitoring Memory Consumption

You can use the Memory tab of the JConsole interface to monitor memory consumption. This tab provides information on memory consumption and memory pools. Figure 2–7 shows the Memory tab.

Figure 2-7 The Memory Tab of the JConsole Interface



The chart on the Memory tab shows Oracle JVM memory usages versus time, for the whole memory space and also for specific memory pools. The memory pools available for Oracle JVM reflect the internal organization of Oracle JVM and correspond to object memories of Oracle JVM Memory Manager. The available memory pools in this release of Oracle Database are:

- New Generation Space

  This is the memory pool from which memory is initially allocated for most objects. This pool is also referred to as the Eden Space.

- Old Generation Space

  This memory pool contains objects that have survived the garbage collection process in Eden Space. This pool is also referred to as the Survival Space.

- Malloc/Free Space

  This memory pool contains objects for which memory is allocated and freed in malloc/free fashion.

- End of Migration Space

  This memory pool contains objects surviving end-of-session migration.

- Dedicated Session Space

  This memory pool is used to allocate memory to session objects in Oracle Dedicated Sessions mode.

- Paged Session Space

  This memory pool is used to allocate memory to session objects that are big and paged.

- Run space

  This memory pool is used to allocate memory to temporary and auxiliary objects.

- Stack space

  This memory pool is used to allocate memory to temporary objects for which memory is allocated and freed in stack-like fashion.

The Details area in the Memory tab displays current memory matrixes that include the following:

- Used

  This matrix indicates the amount of memory currently used by the process running the session.

- Committed

  This matrix indicates the amount of memory guaranteed to be available for use by Oracle JVM, as if the memory has already been allocated. The amount of Committed memory may change over time. But Committed memory will always be greater than or equal to Used memory.

- Max

  This matrix indicates the maximum amount of memory that can be used for memory management. It usually corresponds to the initial configuration of Oracle JVM.

The bar chart at the lower right corner of the Memory tab shows memory consumed by the individual memory pools. The bar turns red when the memory used exceeds the memory usage threshold. You can set the memory usage threshold through an attribute of the `MemoryMXBean`.

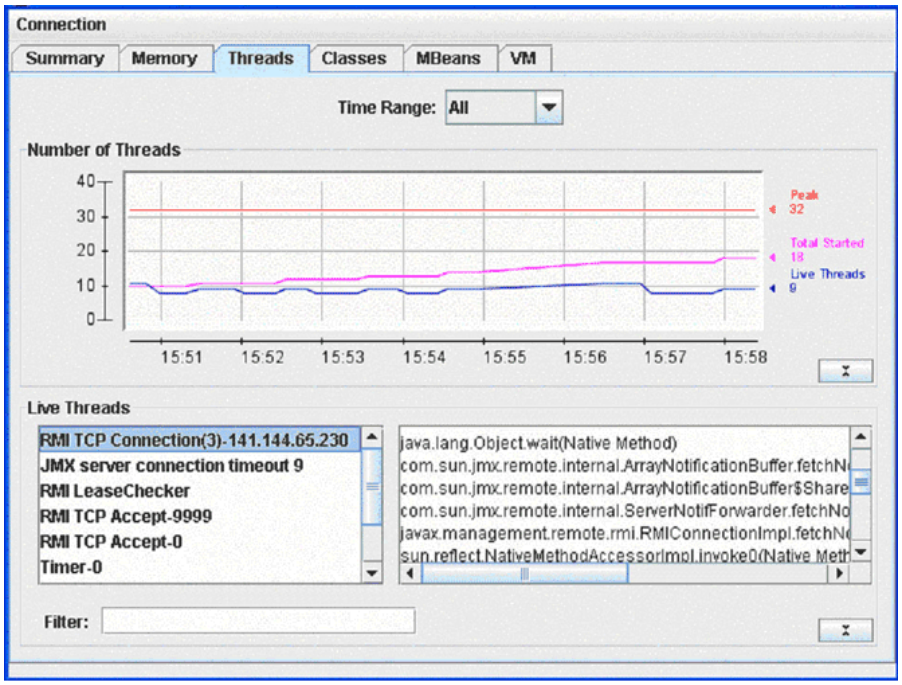
Related Topics

- Memory Thresholds

2.11.5.5 About Monitoring Thread Use

You can use the Threads tab of the JConsole interface to monitor thread usage.

Figure 2-8 The Threads Tab of the JConsole Interface



The chart on the Threads tab displays the number of live threads versus time, with a particular color representing a particular type of thread:

- Magenta signifies total number of threads
- Red signifies peak number of threads
- Blue signifies number of live threads

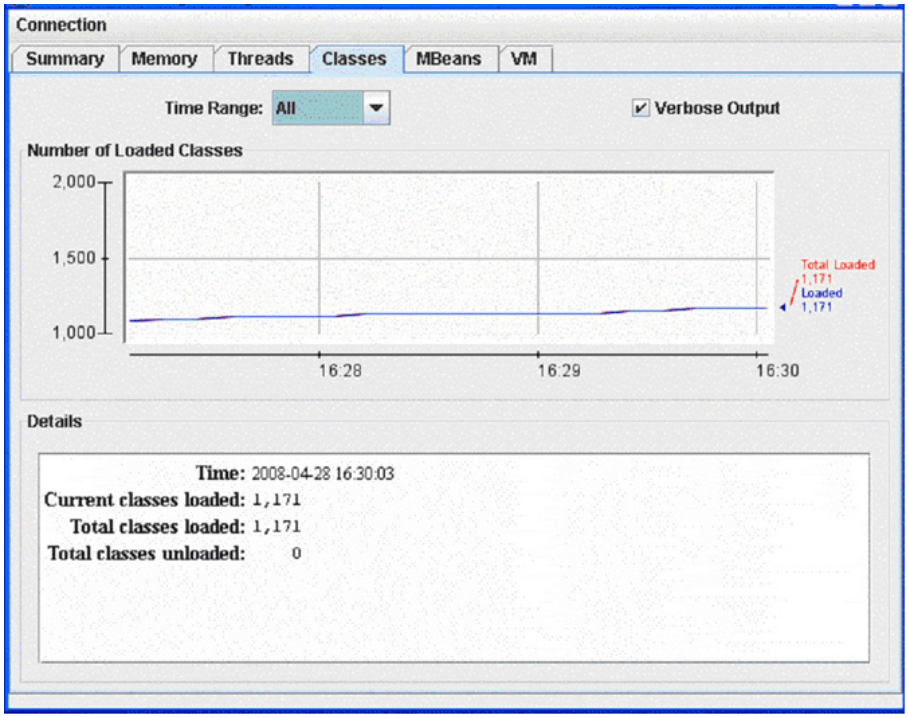
The list of threads on this tab displays the active threads. Select a thread in the list to display information about that thread on the right pane. This information includes name, state, and stack trace of the thread.

The Filter field helps to narrow the threads.

2.11.5.6 About Monitoring Class Loading

You can use the Classes tab of the JConsole interface to monitor class loading. The chart on this tab plots the number of classes loaded versus time.

Figure 2-9 The Classes tab of the JConsole interface

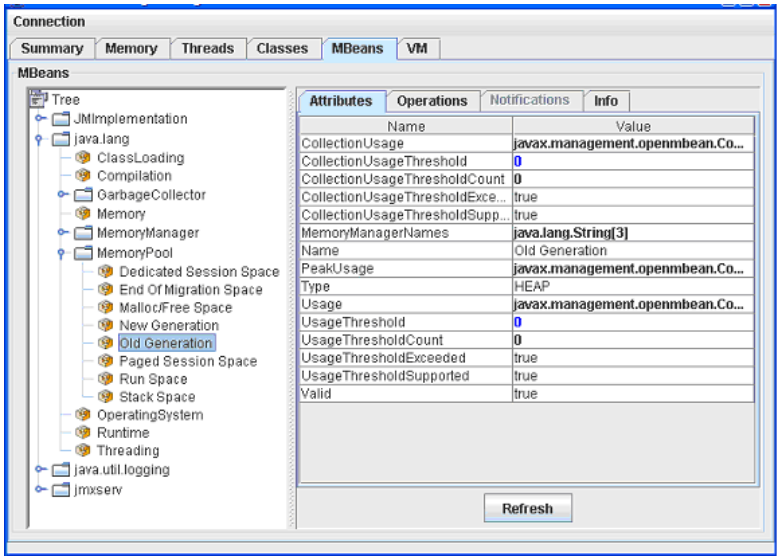


2.11.5.7 About Monitoring and Managing MBeans

You can use the MBeans tab to monitor and manage MBeans. This tab displays information on all the MBeans registered with the platform MBean server.

The tree on the left pane of the MBean tab is called the MBean tree and it shows all the MBeans, organized according to their object Names. When you select an MBean in the MBean tree, then its attributes, operations, notifications, and other information are displayed on the right pane. For example, in Figure 2-10, we have selected the `Old Generation` MemoryPool MBean in the MBean tree on the left and the attributes of the `Old Generation` MemoryPool MBean are displayed on the right.

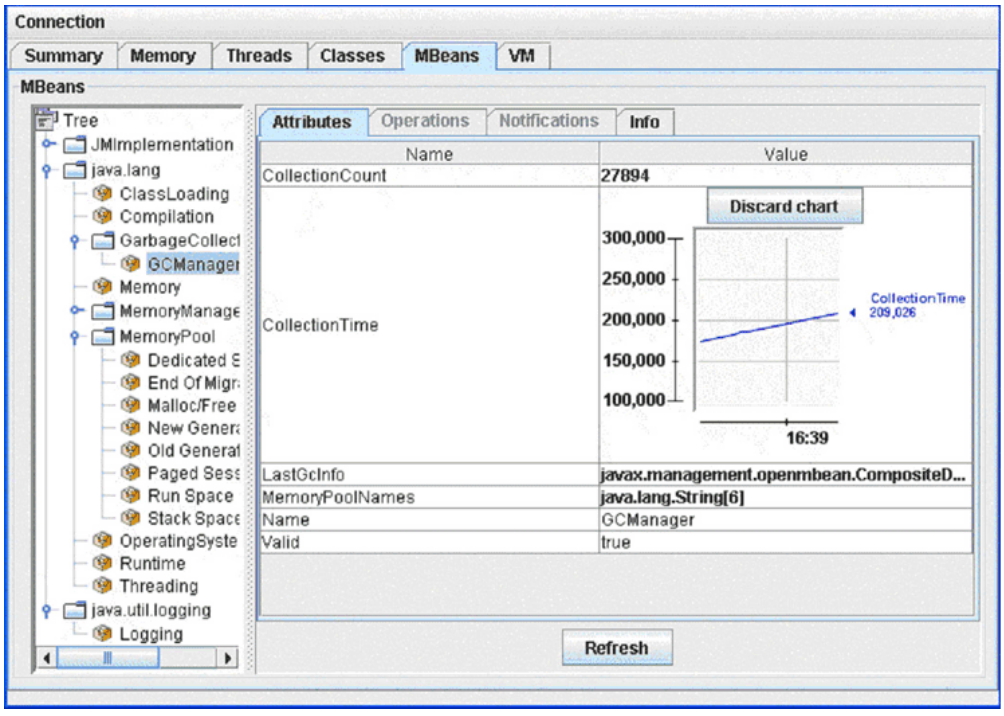
Figure 2-10 Displaying the Attributes of an MBean



You can set the values of an attribute, if it is writeable. The writeable values are displayed in blue color. For example, in Figure 2-10, the attributes `CollectionUaageThreshold` and `UsageThreshold` are writable.

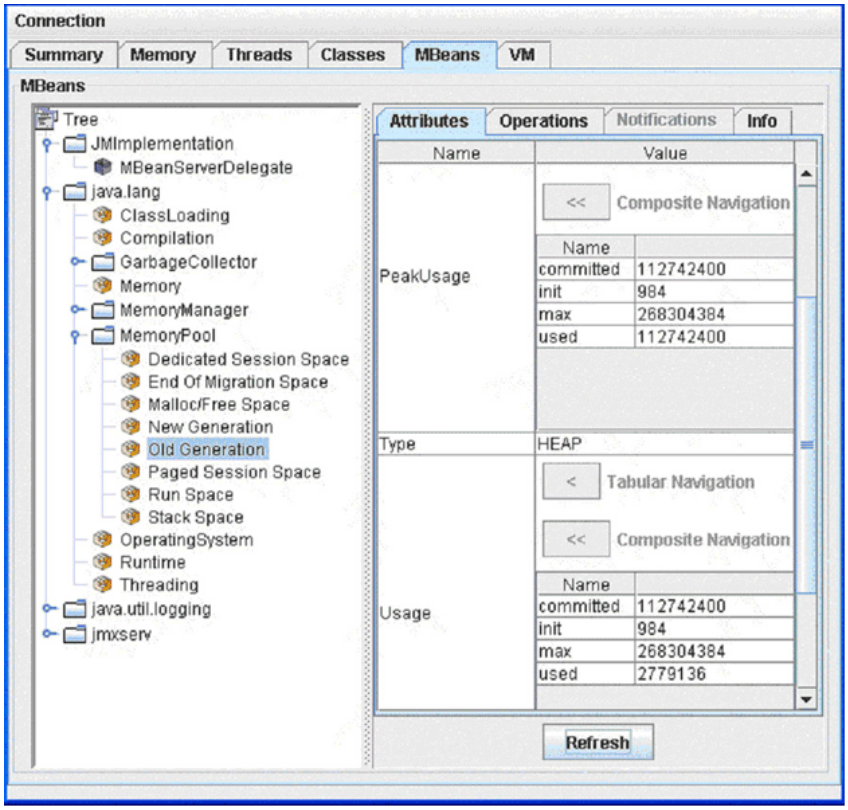
You can also display a chart of the values of an attribute versus time, by double-clicking on the attribute value. For example, if you click on the value of the `CollectionTime` property of the `GCManager` MBean, then you will see a chart similar to Figure 2-11:

Figure 2-11 Displaying a Chart of the Values of an Attribute



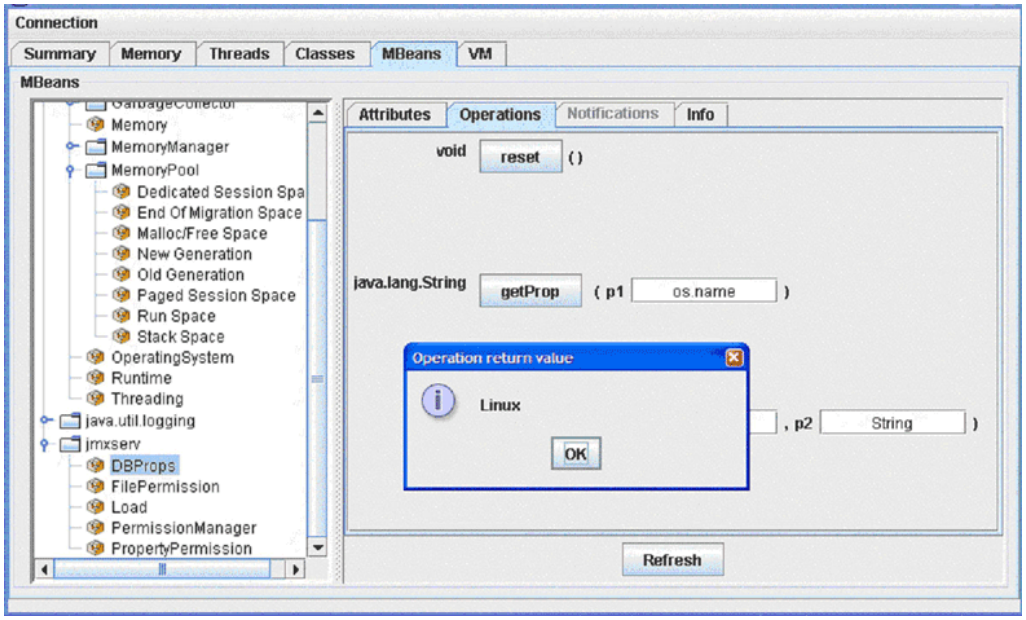
You can display the details of a complex attribute by clicking on the attribute. For example, you can display the details of `Usage` and `PeakUsage` attributes of the Memory Pools as shown in Figure 2-12:

Figure 2-12 Displaying Details of a Complex Attribute in the MBeans Tab



The Operations tab of an MBean provides manageability interface. For example, garbage collection can be invoked on a Memory Pool or Memory Manager by clicking **Perform Garbage Collection**. The JMX demo of Oracle JVM, namely, `javavm/demo/jmx/`, provides several additional custom MBeans that are loaded into Oracle JVM. Here is an example shows the result of the `getProp` operation of the `DBProps` Mbean:

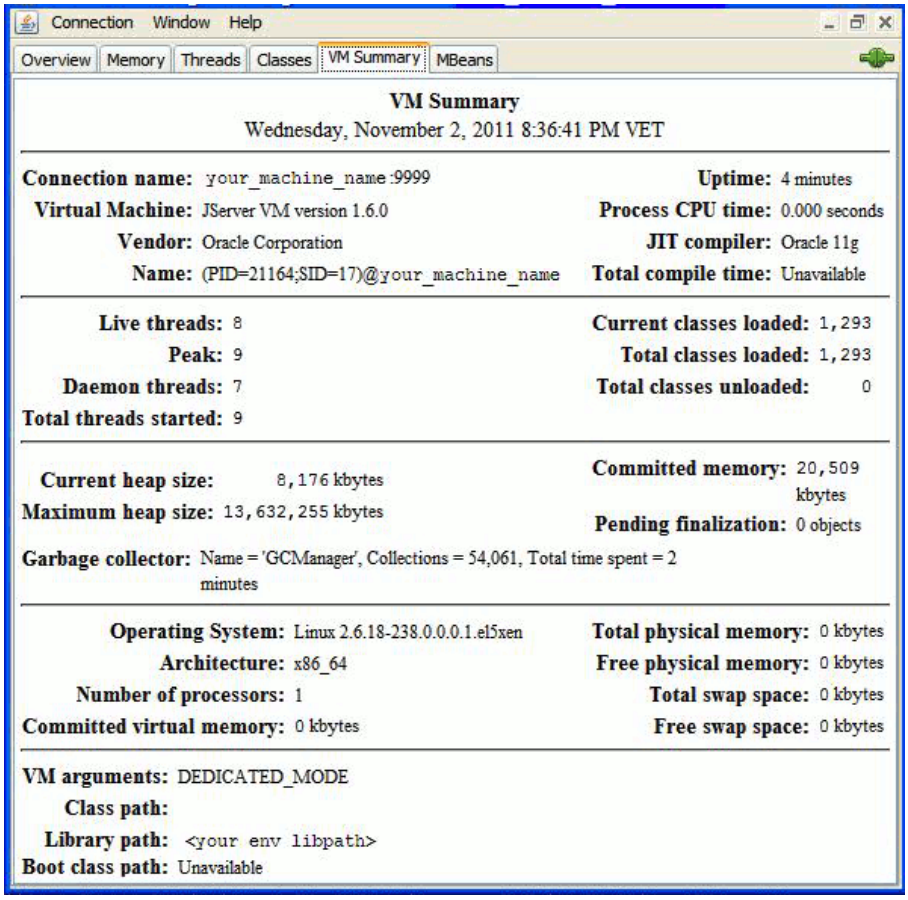
Figure 2-13 Operations Tab of the MBeans Tab of the JConsole Interface



2.11.5.8 About Viewing VM Information

You can use the VM tab of the JConsole interface to view VM information.

Figure 2-14 The VM Tab of the JConsole Interface

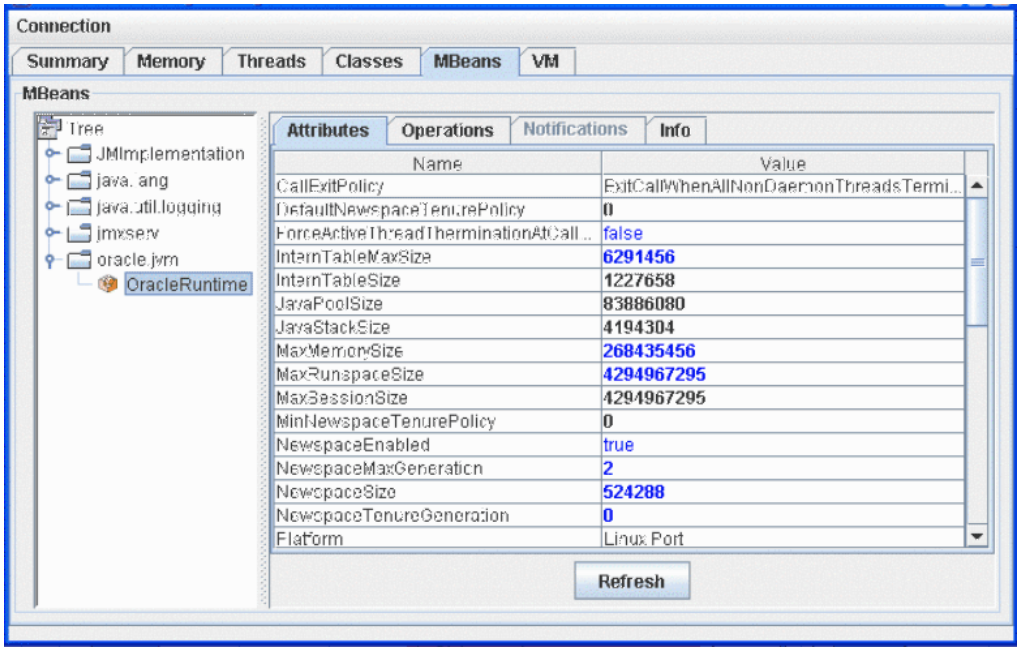


2.11.5.9 The OracleRuntime MBean

When the `dbms_java.start_jmx_agent` procedure is called, then `OracleRuntime` is added to the list of Oracle JVM platform MBeans. This MBean is specific to Oracle JVM.

The Attributes Tab of the `OracleRuntime` MBean exposes most of the parameters manipulated by the `oracle.aurora.vm.OracleRuntime` class. Figure 2-15 shows the Attributes tab of the `OracleRuntime` MBean.

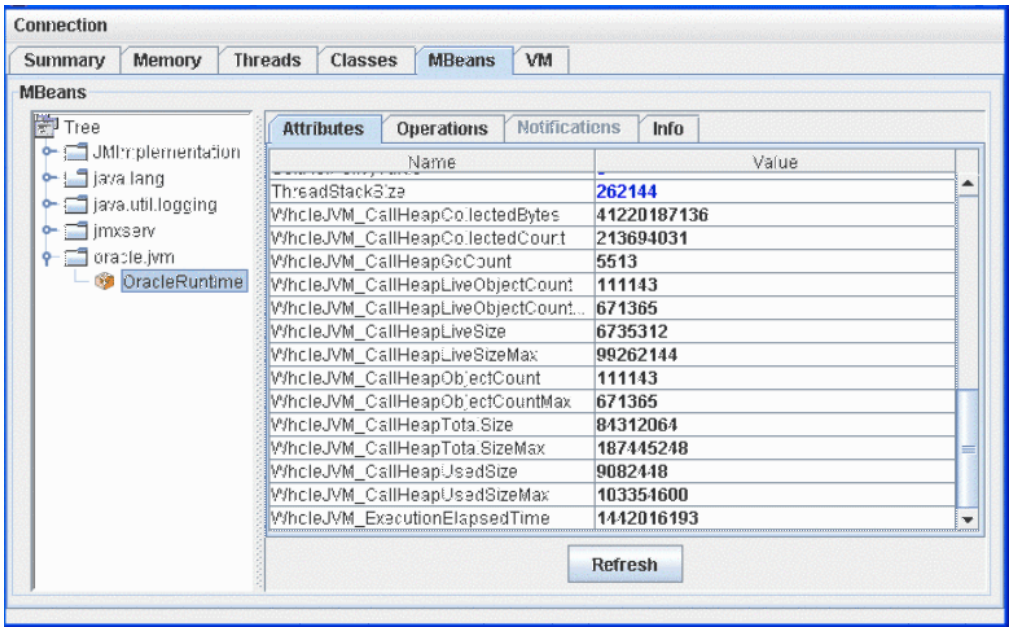
Figure 2-15 Attributes Tab of the OracleRuntime MBean



The parameters displayed in black color are read-only and cannot be modified. For example, `JavaPoolSize`, `Platform`, and so on. Values in blue color are read/write, which means you can modify them. Most of the attributes of the `OracleRuntime` MBean are local to the current session.

The `WholeJVM_` attributes of the `OracleRuntime` MBean are global. These attributes reflect the totals of Oracle JVM memory usage statistics across all Java-enabled sessions in the Database instance, as gathered from the `v$session` and `v$sesstat` performance views. Figure 2-16 displays the `WholeJVM_` attributes of the `OracleRuntime` MBean.

Figure 2-16 OracleRuntime MBean



The Operations Tab of the `OracleRuntime` MBean exposes many of the operations of the `oracle.aurora.vm.OracleRuntime` class.

In addition, individual memory consumption statistics of a specific Java-active Database session can be monitored using the `sessionsRunningJava` and `sessionDetailsBySID` operations as shown in Figure 2-17 and Figure 2-18.

Figure 2-17 Operation sessionsRunningJava

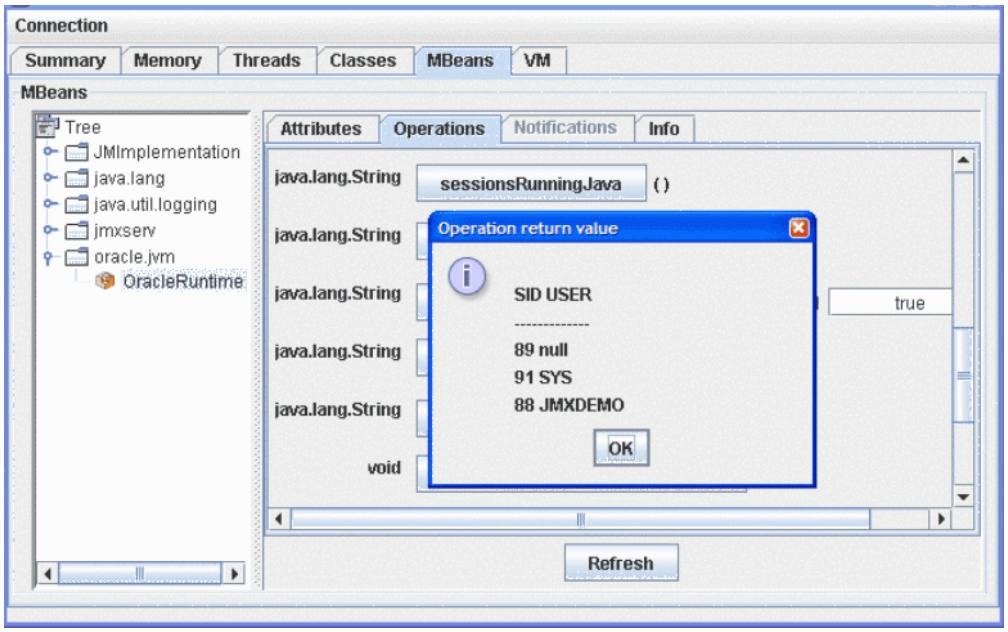
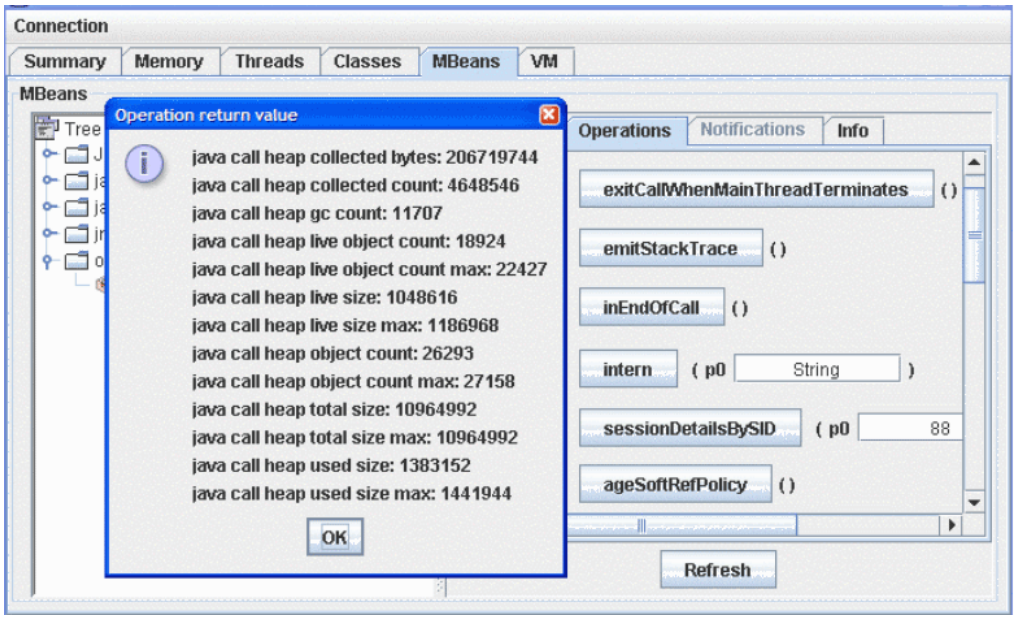


Figure 2-18 Operation sessionDetailsBySID

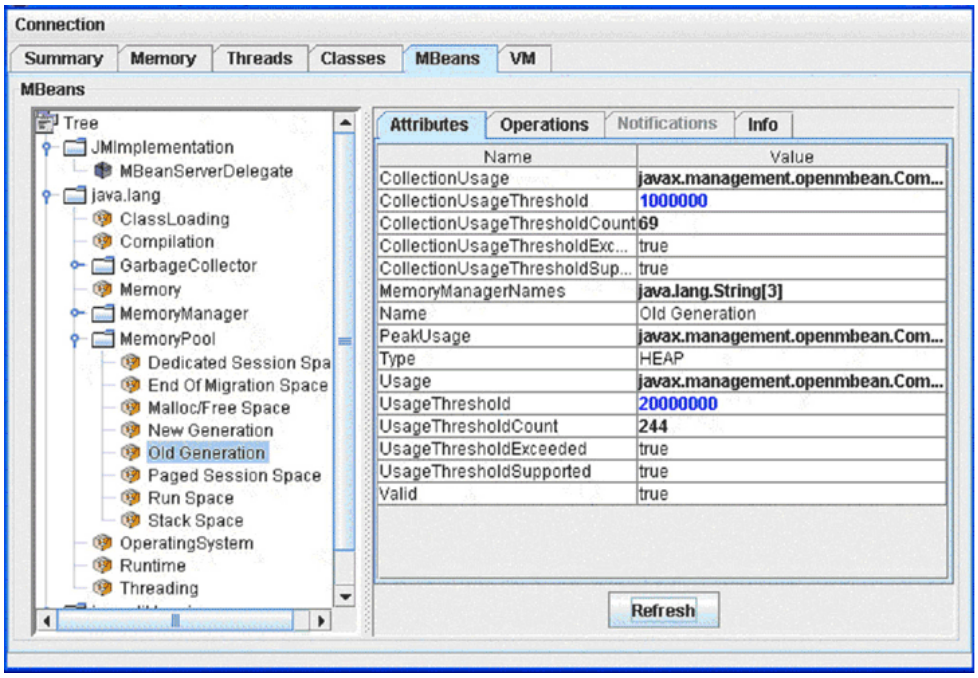


2.11.5.10 Memory Thresholds

The usage threshold is a manageable attribute of the memory pools. Collection usage threshold is a manageable attribute of some of the garbage-collected memory pools. You can set each of these to a positive value to enable corresponding threshold checking for a pool. Setting a threshold to zero disables the threshold checking for the memory pool. By default, threshold checking for all Oracle JVM pools is disabled.

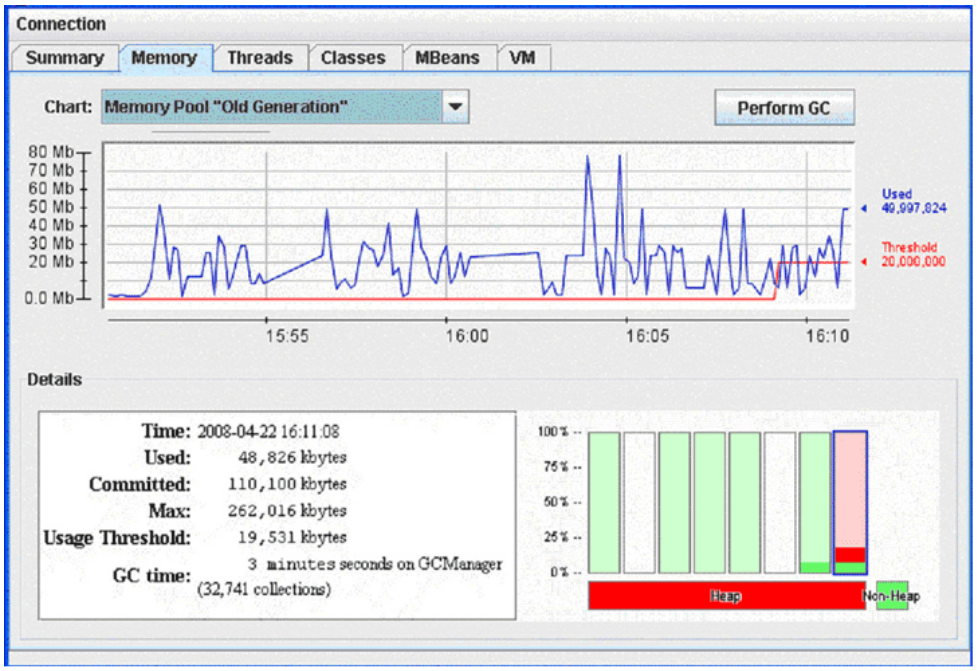
The usage threshold and the collection usage threshold are set in the MBeans tab. For example, if you select the Old Generation memory pool from the tree on the left pane, and set the usage threshold of this memory pool to 20 megabytes and the collection threshold to 1 megabyte, then after a while, the threshold counts will show the number of threshold crossing events as shown in Figure 2-19:

Figure 2-19 Setting the Usage Threshold and Collection Usage Threshold in the MBeans Tab



When the memory usage of the Old Generation memory pool exceeds 20 megabytes, then part of the bar representing the Old Generation memory pool in the JConsole interface turns red. The red portion indicates the portion of used memory that exceeds the usage threshold. The bar representing the heap memory also turns red as shown in Figure 2-20:

Figure 2-20 Memory Tab of the JConsole Interface When Used Memory Exceeds the Usage Threshold



2.11.6 Important Security Notes

By starting the remote listener with disabled SSL and authentication you violate the general security guidelines and hence make server vulnerable to attacks. Therefore, it is always advisable not to use such mode in production environment. This mode is supported for compatibility with JDK and for development; any production use of JMX in Oracle JVM must use secure JMX connections.

When supplying security-related property values to `dbms_java.set_property`, `System.setProperty`, or `dbms_java.start_jmx_agent`, use a non-echo listener or invoke these through an encrypted JDBC connection from a secure application layer, such as Oracle Application Server. Do not store passwords in clear-text files. Use Oracle Wallet to create and manage certificates. Use client certificates for SSL authentication for better security.

> **See Also:**
>
> *Oracle Database Security Guide*

2.11.7 Shared Server Limitations for JMX

On dedicated mode servers, JMX connectivity is supported for the duration of a session. Shared server JMX connectivity is typically limited to a single call. The main factor causing this limitation is the fact that JMX connectivity intrinsically depends on operating system resources such as threads and sockets. These resources do not survive shared server call boundaries. As the result, JMX connectivity is fully supported only for the duration of a single call.

> **Note:**
>
> This restriction only affects agent connectivity and not the state of the MBeanSrver and Mbeans registered in it. The state of the MBeanSrver and Mbeans, and in particular, the statistics, are persevered across shared server call boundaries.

If using dedicated server mode is not feasible, you can still establish JMX connectivity and monitor shared servers by following these guidelines:

- Plan for all JMX management and monitoring activities to happen within a single Java call.
- Do not set the `com.sun.management.jmxremote.port` property by calling the `DBMS_JAVA.set_property` function and do not use the `DBMS_JAVA.start_jmx_agent` method because these calls activate JMX and introduce a shared server call boundary. Instead, start the JMX agent by calling the `oracle.aurora.rdbms.JMXAgent.startOJVMAgent` method directly from the Java code to be monitored. The value for the `com.sun.management.jmxremote.port` property should be passed to the `startOJVMAgent` method. JMX-related properties other than the `com.sun.management.jmxremote.port` property do not wake up a JMX Agent and can be set using any means.

Related Topics

- Shared Servers Considerations

2.12 Overview of Threading in Oracle Database

Oracle JVM is based on the database session model, which is a single-client, nonpreemptive threading model. Although Java in Oracle Database allows running threaded programs, it is single-threaded at the execution level. In this model, JVM runs all Java threads associated with a database session on a single operating system thread. Once dispatched, a thread continues execution until it explicitly yields by calling `Thread.yield()`, blocks by calling `Socket.read()`, or is preempted by the execution engine. Once a thread yields, blocks or is preempted, JVM dispatches another thread.

Oracle JVM has added the following features for better performance and thread management:

- System calls are at a minimum. Oracle JVM has exchanged some of the standard system calls with nonsystem solutions. For example, entering a monitor-synchronized block or method does not require a system call.
- Deadlocks are detected.
  - Oracle JVM monitors for deadlocks between threads. If a deadlock occurs, then Oracle JVM terminates one of the threads and throws the `oracle.aurora.vm.DeadlockError` exception.
  - Single-threaded applications cannot suspend. If the application has only a single thread and you try to suspend it, then the `oracle.aurora.vm.LimboError` exception is thrown.

2.12.1 Thread Life Cycle

In a single-threaded application, a call ends when one of the following events occurs:

- The thread returns to its caller.

- An exception is thrown and is not caught in Java code.
- The `System.exit()`, `OracleRuntime.exitSession()`, or `oracle.aurora.vm.OracleRuntime.exitCall()` method is called.
- The `DBMS_JAVA.endsession()` or `DBMS_JAVA.endsession_and_related_state()` method is called.

If the initial thread creates and starts other Java threads, then the call ends in one of the following ways:

- The main thread returns to its caller or an exception is thrown and not caught in this thread and in either case all other non-daemon threads are processed. Non-daemon threads complete either by returning from their initial method or because an exception is thrown and not caught in the thread.
- Any thread calls the `System.exit()`, `OracleRuntime.exitSession()`, or `oracle.aurora.vm.OracleRuntime.exitCall()` method.
- A call to `DBMS_JAVA.endsession()` or `DBMS_JAVA.endsession_and_related_state()` method.

The following PL/SQL functions in the `DBMS_JAVA` package ensures that when a call ends because of a call to `System.exit()` or `oracle.aurora.vm.OracleRuntime.exitCall()` methods, Oracle JVM does not end the call abruptly or terminate all threads, whether in the dedicated mode or the shared server mode:

- `FUNCTION endsession RETURN VARCHAR2;`
- `FUNCTION endsession_and_related_state RETURN VARCHAR2;`

During a call, a Java program can recursively cause more Java code to be run. For example, your program can issue a SQL query using JDBC that, in turn, calls a trigger written in Java. All the preceding remarks regarding call lifetime apply to the top-most call to Java code, not to the recursive call. For example, a call to `System.exit()` from within a recursive call exits the entire top-most call to Java, not just the recursive call.

Related Topics

- Operating System Resources Affected Across Calls
  In the shared server mode, Oracle JVM closes any open operating system resources at the end of a database call.
- Two-Tier Duration for Java Session State

2.12.2 System.exit(), OracleRuntime.exitSession(), and OracleRuntime.exitCall()

The `System.exit()` method terminates JVM, preserving no Java state. It does not cause the database session to terminate or the client to disconnect.

However, the database session may, and often does, terminate itself immediately afterward. `OracleRuntime.exitSession()` also terminates JVM, preserving no Java state. However, it also terminates the database session and disconnects the client.

The behavior of `OracleRuntime.exitCall()` varies depending on `OracleRuntime.threadTerminationPolicy()`. This method returns a `boolean` value. If this value is `true`, then any active thread should be terminated, rather than left quiescent, at the end of a database call.

- In a shared server process, `threadTerminationPolicy()` is always `true`.

- In a shadow (dedicated) process, the default value is `false`. You can change the value by calling `OracleRuntime.setThreadTerminationPolicy()`.
  - If you set the value to false, that is the default value, all threads are left quiescent but receive a `ThreadDeath` exception for graceful termination.
  - If the value is true, all threads are terminated abruptly.

In addition, there is another method, `OracleRuntime.callExitPolicy()`. This method determines when a call is exited if none of the `OracleRuntime.exitSession()`, `OracleRuntime.exitCall()`, or `System.exit()` methods were ever called. The call exit policy can be set to one of the following, using `OracleRuntime.setCallExitPolicy()`:

- `OracleRuntime.EXIT_CALL_WHEN_MAIN_THREAD_TERMINATES`

  If set to this value, then as soon as the main thread returns or an uncaught exception occurs on the main thread, all remaining threads, both daemon and non-daemon are:
  - Stopped, if `threadTerminationPolicy()` is true, always in shared server mode.
  - Left quiescent, if `threadTerminationPolicy()` is false.
- `OracleRuntime.EXIT_CALL_WHEN_ALL_NON_DAEMON_THREADS_TERMINATE`

  This is the default value. If this value is set, then the call ends when only daemon threads are left running. At this point:
  - If the `threadTerminationPolicy()` is `true`, always in shared server mode, then the daemon threads are stopped.
  - If the `threadTerminationPolicy()` is `false`, then the daemon threads are left quiescent until the next call. This is the default setting for shadow (dedicated) server mode.
- `OracleRuntime.EXIT_CALL_WHEN_ALL_THREADS_TERMINATE`

  If set to this value, then the call ends only when all threads have either returned or ended due to an uncaught exception. At this point, the call ends regardless of the value of `threadTerminationPolicy()`.

2.13 Shared Servers Considerations

> **Note:**
>
> Oracle recommends dedicated servers for performance reasons. Additionally, dedicated servers support a class of applications that rely on threads and sockets that stay open across calls. For example, the JMX agent connectivity functionality.

For sessions that use shared servers, certain limitations exist across calls. The reason is that a session that uses a shared server is not guaranteed to connect to the same process on a subsequent database call, and hence the session-specific memory and objects that need to live across calls are saved in the SGA. This means that process-specific resources, such as threads, open files, and sockets, must be cleaned up at the end of each call, and therefore, will not be available for the next call.

This section covers the following topics:

- End-of-Call Migration

- Oracle-Specific Support for End-of-Call Optimization
- The EndOfCallRegistry.registerCallback() Method
- The EndOfCallRegistry.runCallbacks() Method
- The Callback Interface
- The Callback.act() method
- Operating System Resources Affected Across Calls

Related Topics

- Managing Your Applications Using JMX

2.13.1 End-of-Call Migration

In the shared server mode, Oracle Database preserves the state of your Java program between calls by migrating all objects that are reachable from `static` variables to session space at the end of the call. Session space exists within the session of the client to store `static` variables and objects that exist between calls. Oracle JVM automatically performs this migration operation at the end of every call.

This migration operation is a memory and performance consideration. Hence, you should be aware of what you designate to exist between calls and keep the `static` variables and objects to a minimum. If you store objects in `static` variables needlessly, then you impose an unnecessary burden on the memory manager to perform the migration and consume per-session resources. By limiting your `static` variables to only what is necessary, you help the memory manager and improve the performance of your server.

To maximize the number of users who can run your Java program at the same time, it is important to minimize the footprint of a session. In particular, to achieve maximum scalability, an inactive session should take up as little memory space as possible. A simple technique to minimize footprint is to release large data structures at the end of every call. You can lazily re-create many data structures when you need them again in another call. For this reason, Oracle JVM has a mechanism for calling a specified Java method when a session is about to become inactive, such as at the end of a call.

This mechanism is the `EndOfCallRegistry` notification. It enables you to clear `static` variables at the end of the call and reinitialize the variables using a lazy initialization technique when the next call comes in. You should run this only if you are concerned about the amount of storage you require the memory manager to store in between calls. It becomes a concern only for complex stateful server applications that you implement in Java.

The decision of whether to null-out data structures at the end of the call and then re-create them for each new call is a typical time and space trade-off. There is some extra time spent in re-creating the structure, but you can save significant space by not holding on to the structure between calls. In addition, there is a time consideration, because objects, especially large objects, are more expensive to access after they have been migrated to session space. The penalty results from the differences in representation of session, as opposed to objects based on call-space.

Examples of data structures that are candidates for this type of optimization include:

- Buffers or caches.
- Static fields, such as arrays, which once initialized can remain unchanged during the course of the program.
- Any dynamically built data structure that can have a space-efficient representation between calls and a more speed-efficient representation for the duration of a call. This can

be tricky and may complicate your code, making it hard to maintain. Therefore, you should consider doing this only after demonstrating that the space saved is worth the effort.

2.13.2 Oracle-Specific Support for End-of-Call Optimization

You can register the `static` variables that you want cleared at the end of the call when the buffer, field, or data structure is created. Within the `oracle.aurora.memoryManager.EndOfCallRegistry` class, the `registerCallback()` method takes an object that implements a `Callback` object. The `registerCallback()` method stores this object until the end of the call. At the end of the call, Oracle JVM calls the `act()` method within all registered `Callback` objects. The `act()` method within the `Callback` object is implemented to clear the user-defined buffer, field, or data structure. Once cleared, the `Callback` object is removed from the registry.

> **Note:**
>
> If the end of the call is also the end of the session, then callbacks are not started, because the session space will be cleared anyway.

A weak table holds the registry of end-of-call callbacks. If either the `Callback` object or value are not reachable from the Java program, then both the object and the value will be dropped from the table. The use of a weak table to hold callbacks also means that registering a callback will not prevent the garbage collector from reclaiming that object. Therefore, you must hold on to the callback yourself if you need it, and you cannot rely on the table holding it back.

The way you use `EndOfCallRegistry` depends on whether you are dealing with objects held in `static` fields or instance fields.

Static fields

Use `EndOfCallRegistry` to clear state associated with an entire class. In this case, the `Callback` object should be held in a `private static` field. Any code that requires access to the cached data that was dropped between calls must call a method that lazily creates, or re-creates, the cached data.

Consider the following example:

```
import oracle.aurora.memoryManager.Callback;
import oracle.aurora.memoryManager.EndOfCallRegistry;

class Example
{
  static Object cachedField = null;
  private static Callback thunk = null;

  static void clearCachedField()
  {
    // clear out both the cached field, and the thunk so they don't
    // take up session space between calls
    cachedField = null;
    thunk = null;
  }

  private static Object getCachedField()
  {
    if (cachedField == null)
```

```
    {
      // save thunk in static field so it doesn't get reclaimed
      // by garbage collector
      thunk = new Callback () {
        public void act(Object obj)
        {
          Example.clearCachedField();
        }
      };

      // register thunk to clear cachedField at end-of-call.
      EndOfCallRegistry.registerCallback(thunk);
      // finally, set cached field
      cachedField = createCachedField();
    }
    return cachedField;
  }

  private static Object createCachedField()
  {
    ...
  }
}
```

The preceding example does the following:

1. Creates a `Callback` object within a `static` field, `thunk`.
2. Registers this `Callback` object for end-of-call migration.
3. Implements the `Callback.act()` method to free up all `static` variables, including the `Callback` object itself.
4. Provides a method, `createCachedField()`, for lazily re-creating the cache.

When you create the cache, the `Callback` object is automatically registered within the `getCachedField()` method. At end-of-call, Oracle JVM calls the registered `Callback.act()` method, which frees the static memory.

Instance fields

Use `EndOfCallRegistry` to clear state in data structures held in instance fields. For example, when a state is associated with each instance of a class, each instance has a field that holds the cached state for the instance and fills in the cached field as necessary. You can access the cached field with a method that ensures the state is cached.

Consider the following example:

```
import oracle.aurora.memoryManager.Callback;
import oracle.aurora.memoryManager.EndOfCallRegistry;

class Example2 implements Callback
{
  private Object cachedField = null;

  public voidact (Object obj)
  {
    // clear cached field
    cachedField = null;
    obj = null;
  }

  // our accessor method
```

```
  private static Object getCachedField()
  {
    if (cachedField == null)
    {
      // if cachedField is not filled in then you must
      // register self, and fill it in.
      EndOfCallRegistry.registerCallback(self);
      cachedField = createCachedField();
    }
    return cachedField;
  }

  private Object createCachedField()
  {
    ...
  }
}
```

The preceding example does the following:

1. Implements the instance as a `Callback` object.
2. Implements the `Callback.act()` method to free up the instance fields.
3. When you request a cache, the `Callback` object registers itself for the end-of-call migration.
4. Provides a method, `createCachedField()`, for lazily re-creating the cache.

When you create the cache, the `Callback` object is automatically registered within the `getCachedField()` method. At end-of-call, Oracle JVM calls the registered `Callback.act()` method, which frees the cache.

This approach ensures that the lifetime of the `Callback` object is identical to the lifetime of the instance, because they are the same object.

2.13.3 The EndOfCallRegistry.registerCallback() Method

The `registerCallback()` method installs a `Callback` object within a registry. At the end of the call, Oracle JVM calls the `act()` method of all registered `Callback` objects.

You can register your `Callback` object by itself or with an `Object` instance. If you need additional information stored within an object to be passed into `act()`, then you can register this object with the `value` parameter, which is an instance of `Object`.

The following are the valid signatures of the `registerCallback()` method:

```
public static void registerCallback(Callback thunk, Object value);

public static void registerCallback(Callback thunk);
```

The following table lists the parameters of `registerCallback` and their description:

Parameter	Description
`thunk`	The `Callback` object to be called at the end-of-call migration.
`value`	If you need additional information stored within an object to be passed into `act()`, then you can register this object with the `value` parameter. In some cases, the `value` parameter is necessary to hold the state that the callback needs. However, most users do not need to specify a value for this parameter.

2.13.4 The EndOfCallRegistry.runCallbacks() Method

The signature of the `runCallbacks()` method is as follows:

```
static void runCallbacks()
```

JVM calls this method at end-of-call and calls `act()` for every `Callback` object registered using `registerCallback()`. It is called at end-of-call, before object migration and before the last finalization step.

> **Note:**
>
> Do not call this method in your code.

2.13.5 The Callback Interface

The interface is declared as follows:

```
Interface oracle.aurora.memoryManager.Callback
```

Any object you want to register using `EndOfCallRegistry.registerCallback()` must implement the `Callback` interface. This interface can be useful in your application, where you require notification at end-of-call.

2.13.6 The Callback.act() method

The signature of the `act()` method is as follows:

```
public void act(Object value)
```

You can implement any activity that you require to occur at the end of the call. Usually, this method contains procedures for clearing any memory that would be saved to session space.

2.13.7 Operating System Resources Affected Across Calls

In the shared server mode, Oracle JVM closes any open operating system resources at the end of a database call.

The following table lists the operating system resources across calls and specifies the corresponding lifetime.

Resource	Lifetime
Files	The system closes all files left open when a database call ends.
Threads	All threads are terminated when a call ends.
Sockets	• Client sockets can exist across calls. • Server sockets terminate when the call ends.

Resource	Lifetime
Objects that depend on operating system resources	Regardless of the usable lifetime of the object, the Java object can be valid for the duration of the session. This can occur, for example, if the Java object is stored in a `static` class variable, or a class variable references it directly or indirectly. If you attempt to use one of these Java objects after its usable lifetime is over, then Oracle Database will throw an exception. This is true for the following examples: • If an attempt is made to read from a `java.io.FileInputStream` that was closed at the end of a previous call, then a `java.io.IOException` is raised. • `java.lang.Thread.isAlive()` is `false` for any `Thread` object running in a previous call and still accessible in a subsequent call.

You should close resources that are local to a single call when the call ends. However, for `static` objects that hold on to operating system resources, you must be aware of how these resources are affected after the call ends.

Files

In the shared server mode, Oracle JVM automatically closes open operating system constructs when the call ends. This can affect any operating system resources within your Java object. If you have a file opened within a `static` variable, then the file handle is closed at the end of the call for you. Therefore, if you hold on to the `File` object across calls, then the next usage of the file handle throws an exception.

In the following example, the `Concat` class enables multiple files to be written into a single file, `outFile`. On the first call, `outFile` is created. The first input file is opened, read, written to `outFile`, and the call ends. Because `outFile` is defined as a `static` variable, it is moved into session space between call invocations. However, the file handle is closed at the end of the call. The next time you call `addFile()`, you will get an exception.

Example 2-5 Compromising Your Operating System Resources

```
public class Concat
{
  static File outFile = new File("outme.txt");
  FileWriter out = new FileWriter(outFile);

  public static void addFile(String[] newFile)
  {
    File inFile = new File(newFile);
    FileReader in = new FileReader(inFile);
    int i;

    while ((i = in.read()) != -1)
      out.write(i);
    in.close();
  }
}
```

There are workarounds. To ensure that your handles stay valid, close your files, buffers, and so on, at the end of every call, and reopen the resource at the beginning of the next call. Another option is to use the database rather than using operating system resources. For example, try to use database tables rather than a file. Alternatively, do not store operating system resources within `static` objects that are expected to live across calls. Instead, use operating system resources only within objects local to the call.

The following example shows how you can perform concatenation, without compromising your operating system resources. The `addFile()` method opens the `outme.txt` file within each call, ensuring that anything written into the file is appended to the end. At the end of each call, the file is closed. Two things occur:

- The `File` object no longer exists outside a call.
- The operating system resource, the `outme.txt` file, is reopened for each call. If you had made the `File` object a `static` variable, then the closing of `outme.txt` within each call would ensure that the operating system resource is not compromised.

Example 2-6 Correctly Managing Your Operating System Resources

```
public class Concat
{

  public static void addFile(String[] newFile)
  {
    /*open the output file each call; make sure the input*/
    /*file is written out to the end by making it "append=true"*/
    FileWriter out = new FileWriter("outme.txt", TRUE);
    File inFile = new File(newFile);
    FileReader in = new FileReader(inFile);
    int i;

    while ((i = in.read()) != -1)
      out.write(i);
    in.close();

    /*close the output file between calls*/
    out.close();
  }
}
```

Sockets

Sockets are used in setting up a connection between a client and a server. For each database connection, sockets are used at either end of the connection. Your application does not set up the connection. The connection is set up by the underlying networking protocol, TTC or IIOP of Oracle Net.

> **See Also:**
>
> "Configuring Oracle JVM" for information about how to configure your connection.

You may also want to set up another connection, for example, connecting to a specified URL from within one of the classes stored within the database. To do so, instantiate sockets for servicing the client and server sides of the connection using the following:

- The `java.net.Socket()` constructor creates a client socket.
- The `java.net.ServerSocket()` constructor creates a server socket.

A socket exists at each end of the connection. The server side of the connection that listens for incoming calls is serviced by a `ServerSocket` instance. The client side of the connection that sends requests is serviced through a `Socket` instance. You can use sockets as defined within JVM with the restriction that a `ServerSocket` instance within a shared server cannot exist across calls.

The following table lists the socket types and their description:

Socket Type	Description
Socket	Because the client side of the connection is outbound, the `Socket` instance can be serviced across calls within a shared server.
ServerSocket	The server side of the connection is a listener. The `ServerSocket` instance is closed at the end of a call within a shared server. The shared servers move on to another client at the end of every call. You will receive an I/O exception stating that the socket was closed, if you try to use the `ServerSocket` instance outside of the call it was created in.

Threads

In the shared server mode, when a call ends because of a return or uncaught exceptions, Oracle JVM throws `ThreadDeathException` in all daemon threads. `ThreadDeathException` essentially forces threads to stop running. Code that depends on threads living across calls does not behave as expected in the shared server mode. For example, the value of a `static` variable that tracks initialization of a thread may become incorrect in subsequent calls because all threads are stopped at the end of a database call.

As a specific example, the standard RMI Server functions in the shared server mode. However, it is useful only within the context of a single call. This is because the RMI Server forks daemon threads, which are in the shared server mode, are stopped at the end of call, that is, the daemon thread are stopped when all non-daemon threads return. If the RMI server session is reentered in a subsequent call, then these daemon threads are not restarted and the RMI server fails to function properly.

2.14 Oracle JVM Rolling Patching

Oracle JVM Patching is the process of patching the Oracle executable and the corresponding Java classes that make up the Oracle JVM, where both must be in sync to run Java in the database.

Oracle JVM uses rolling patching, which enables one instance to continue running the current version of the JVM classes with the current executable and customer Java classes, while the other instances (both the Oracle executable and the JVM classes) are being patched. Once patched, an instance with the new executable and JVM classes are able to immediately run customer Java code as soon as it is started.

> **See Also:**
>
> The following MoS Note for more information https://support.oracle.com/rs?type=doc&id=2217053.1