

NEAR EAST UNIVERSITY



Faculty of Engineering

Department Of Computer Engineering

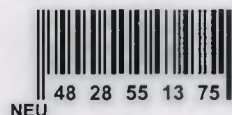
**DEVELOPMENT AND INTEGRATION OF DBMS
WITH ORACLE INTEGRATED INTO JAVA**

**Graduation Project
COM – 400**

Student : Tariq Javed

Supervisor: Mr. Halil Adahan

Nicosia – 2003



ABSTRACT

Database management has evolved from a specialized computer application to a central component of a modern computing environment. As such, database systems have become an essential part of computer science education. Oracle provides a secure platform for database management. The Oracle server is a full-featured RDBMS that is ideally suited to support sophisticated client/server environments. Many features of the Oracle internal architecture are designed to provide high availability, maximum throughput, security, and efficient use of its host's resources. Oracle's Net8 feature provide the Oracle's functionality on the network. The Oracle server can be implemented on the network and fully supports enterprise applications.

The Java technology is an object-oriented, platform-independent, multithreaded, programming environment. It allows us to securely extend our enterprise through platform independence. All kinds of systems can talk to each other regardless of the underlying hardware or system software. We can access Oracle database using JDBC and SQLJ which are rich features of Java to support development of database applications using Java platform. We can give remote access to the applications by using Java's Remote Method Invocation package.



LIST OF FIGURES

Figure1.1 Oracle8i Kernal	2
Figure1.2 Structure of Oracle	3
Figure2.1 Network Listener In a Typical Net8 Connection	39
Figure 2-2 Bequeathed Connection To a Dedicated Server Process	39
Figure2.3 Redirected Connection To a Prespawned Dedicated Server Process	40
Figure2.4 Redirected Connection To a Dispatcher Server Process	41
Figure 2-5 OSI Communications Stack	46
Figure2.6 Typical Communications Stack in an Oracle environment	48
Figure2.7 The Distributed and Nondistributed Models Contrasted	53
Figure2.8 RMI Interfaces and Classes	55

TABLE OF CONTENTS

CHAPTER 1

1.1 RELATIONAL DATABASE MANAGEMENT SYSTEM	1
1.2 RDBMS COMPONENTS	1
1.2.1 The RDBMS Kernel	1
1.2.2 The Data Dictionary	2
1.3 ORACLE DATABASE	4
1.4 ORACLE FILES	4
1.4.1 Database Files	4
1.4.2 Control Files	5
1.4.3 Redo Logs	5
1.4.3.1 Online Redo Logs	6
1.4.3.2 Offline/Archived Redo Logs	6
1.4.4 Other Supporting Files	6
1.5 SYSTEM AND USER PROCESSES	6
1.5.1 Mandatory System Processes	7
1.5.1.1 DBWR (Database Writer)	7
1.5.1.2 LGWR (Log Writer)	7
1.5.1.3 SMON (System Monitor)	7
1.5.1.4 PMON (Process Monitor)	8
1.5.2 Optional System Processes	8
1.5.2.1 ARCH (Archiver)	8
1.5.2.2 CKPT (Checkpoint Process)	9
1.5.2.3 RECO (Recoverer)	9

1.5.2.4 LCK (Lock)	10
1.5.3 User Processes	10
1.5.3.1 Single Task	10
1.5.3.2 Dedicated Server Processes	11
1.5.3.3 The Multi-Threaded Server	11
1.6 ORACLE MEMORY	12
1.6.1 System Global Area (SGA)	12
1.6.1.1 Database Buffer Cache	12
1.6.1.2 Redo Cache	13
1.6.1.3 Shared Pool Area	13
1.6.1.4 SQL Area	13
1.6.1.5 Dictionary Cache	14
1.6.2 Process Global Area	14

CHAPTER 2

2.1 ORACLE ACCESS WITH JDBC	16
2.1.1 Driver Types	17
2.1.1.1 Thin driver	17
2.1.1.2 OCI8 driver	17
2.1.2 The DriverManager Class	17
2.1.3 The Driver Class	18
2.1.4 The Connection Class	18
2.1.5 The Statement Class	19
2.1.6 The ResultSet Class	19
2.2 SQLJ	20
2.2.1 SQLJ Components	21
2.2.1.1 Oracle SQLJ translator	21
2.2.1.2 Oracle SQLJ runtime	21
2.2.1.3 SQLJ Profiles	22
2.2.2 Oracle Extensions to the SQLJ Standard	22
2.2.3 Basic Translation Steps and Runtime Processing	23
2.2.3.1 Translation Steps	23
2.2.3.2 Runtime Processing	24
2.2.4 SQLJ Declarations	25
2.2.5 Java Host Expressions, Context Expressions, and Result Expressions	25
2.2.5.1 Host Expressions	26
2.2.5.2 Context Expressions	26
2.2.5.3 Result Expressions	27
2.2.6 Stored Procedure and Function Calls	27

2.2.7 Multithreading in SQLJ	27
2.2.8 SQLJ and JDBC Interoperability	28
2.2.8.1 Converting from Connection Contexts to JDBC Connections	29
2.2.8.2 Converting from JDBC Connections to Connection Contexts	29
2.2.8.3 Shared Connections	29
2.2.9 SQLJ In the Server	30
2.2.9.1 Creating SQLJ Code for Use within the Server	31
2.2.9.2 Database Connections within the Server	31
2.2.9.3 Coding Issues within the Server	31
2.2.9.4 Name Resolution in the Server	31
2.2.9.5 SQL Names versus Java Names	32
2.2.9.6 Translating SQLJ Source on a Client and Loading Components	33
2.2.9.7 Error Output from the Server Embedded Translator	33
2.3 NET8	
2.3.1 Introduction to Net8	34
2.3.2 Advantages of Net8	34
2.3.2.1 Network Transparency	34
2.3.2.2 Protocol Independence	34
2.3.2.3 Media/Topology Independence	35

2.3.2.4 Heterogeneous Networking	35
2.3.2.5 Large Scale Scalability	35
2.3.3 Net8 Features	35
2.3.3.1 Scalability Features	35
2.3.3.2 Manageability Features	36
2.3.3.2.1 Host Naming	36
2.3.3.2.2 Oracle Net8 Assistant	36
2.3.3.3 Multiprotocol Support Using Oracle Connection Manager	37
2.3.3.4 Oracle Trace Assistant	37
2.3.3.5 Native Naming Adapters	37
2.3.4 Net8 Operations	37
2.3.5 Connect Operations	38
2.3.5.1 Connecting to Servers	38
2.3.5.2 Establishing Connections with the Network Listener	38
2.3.5.2.1 Bequeathed Sessions to Dedicated Server Processes	39
2.3.5.2.2 Redirected Sessions to Existing Server Processes	40
2.3.5.2.3 Refused Sessions	42
2.3.6 Disconnecting from Servers	42
2.3.6.1 User-Initiated Disconnect	42
2.3.6.2 Additional Connection Request	42
2.3.6.3 Abnormal Connection Termination	42
2.3.6.4 Timer Initiated Disconnect or Dead Connection Detection	42

2.3.7 Data Operations	43
2.3.8 Exception Operations	43
2.3.9 Net8 and the Transparent Network Substrate (TNS)	44
2.3.10 Net8 Architecture	44
2.3.10.1 Distributed Processing	44
2.3.10.2 Stack Communications	45
2.3.10.3 Stack Communications in an Oracle networking environment	47
2.3.10.3.1 Client-Server Interaction	47
2.3.10.4 Server-to-Server Interaction	51
2.4 DISTRIBUTED COMPUTING USING JAVA	52
2.4.1 Distributed Object Applications	52
2.4.2 The Distributed and Nondistributed Models Contrasted	53
2.4.3 RMI Interfaces and Classes	54
2.4.3.1 The java.rmi.Remote Interface	55
2.4.4 Parameter Passing in Remote Method Invocation	55
2.4.4.1 Passing Non-remote Objects	55
2.4.4.2 Passing Remote Objects	56
2.4.4.3 Referential Integrity	56
2.4.4.4 Class Annotation	56
2.4.4.5 Parameter Transmission	56
2.4.5 Locating Remote Objects	57
2.4.6 Stubs and Skeletons	58

2.4.7 Thread Usage in Remote Method Invocations	58
2.4.8 Garbage Collection of Remote Objects	59
2.4.9 Dynamic Class Loading	60
2.4.10 RMI Through Firewalls Via Proxies	61
2.4.10.1 How an RMI Call is Packaged within the HTTP Protocol	61
2.4.10.2 The Default Socket Factory	62
2.4.10.3 Configuring the Client	62
2.4.10.4 Configuring the Server	63
2.4.10.5 Performance Issues and Limitations	63
 CHAPTER 3	
3.1 INTRODUCTION	64
3.2 PROGRAM IMPLEMENTATION	64
3.2.1 Database	64
3.2.2 Application	65
 SUMMARY	67
CONCLUSION	68
APPENDICES	
APPENDIX A	69
A.1 Main.java	69
A.2 StudentNew.java	72
A.3 StudentChange.java	76
A.4 StudentDelete.java	81
A.5 StudentView.java	85
APPENDIX B	
GLOSSARY OF JAVA AND RELATED TERMS	90-97
REFERENCES	98

CHAPTER 1

1.1 RELATIONAL DATABASE MANAGEMENT SYSTEM

A database is an integrated collection of related data. Given a specific data item, the structure of a database facilitates the access to data related to it. A relational database is a type of database based in the relational model. A relational database management system is the software that manages a relational database. These systems come in several varieties, ranging from single-user desktop systems to full-featured, global, enterprise-wide systems, such as Oracle8.

1.2 RDBMS COMPONENTS

Two important pieces of an RDBMS architecture are the kernel, which is the software, and the data dictionary, which consists of the system-level data structures used by the kernel to manage the database.

1.2.1 The RDBMS Kernel

You might think of an RDBMS as an operating system or set of subsystems, designed specifically for controlling data access, its primary functions are storing, retrieving, and securing data. Like an operating system, Oracle8i manages and controls access to a given set of resources for concurrent database users. The subsystems of an RDBMS closely resemble those of a host operating system and tightly integrate with the host's services for machine-level access to resources such as memory, CPU, devices, and file structures. An RDBMS such as Oracle8i maintains its own list of authorized users and their associated privileges, manages memory caches and paging, controls locking for concurrent resource usage, dispatches and schedules user requests, and manages space usage within its tablespace structures. Figure 1.1 illustrates the primary subsystems of the Oracle8i kernel that manage the database.

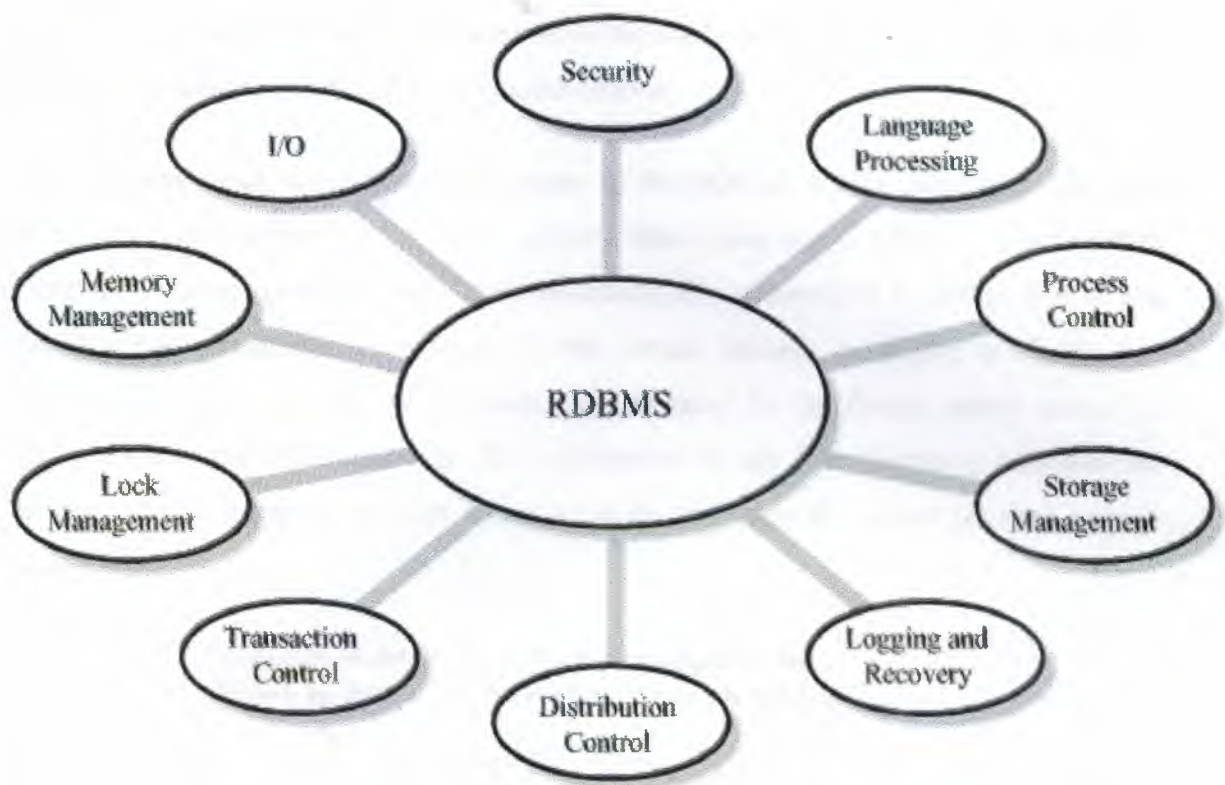


Figure1.1 Oracle8i Kernel

1.2.2 The Data Dictionary

A fundamental difference between an RDBMS and other database and file systems is in the way that they access data. A RDBMS enables you to reference physical data in a more abstract, logical fashion, providing ease and flexibility in developing application code. Programs using an RDBMS access data through a database engine, creating independence from the actual data source and insulating applications from the details of the underlying physical data structures. Rather than accessing a customer number as bytes 1 through 10 of the customer record, an application simply refers to the attribute Customer Number. The RDBMS takes care of where the field is stored in the database. Consider the amount of programming modifications that you must make if you change a record structure in a file system-based application. However, using an RDBMS, the

application code would continue to reference the attribute by name rather than by record position, alleviating the need for any modifications.

This data independence is possible because of the RDBMS's data dictionary. The data dictionary stores meta-data for all the objects that reside in the database. Oracle's data dictionary is a set of tables and database objects that is stored in a special area of the database and maintained exclusively by the Oracle kernel. As shown in Figure 1.2, requests to read or update the database are processed by the Oracle kernel using the information in the data dictionary. The information in the data dictionary validates the existence of the objects, provides access to them, and maps the actual physical storage location.

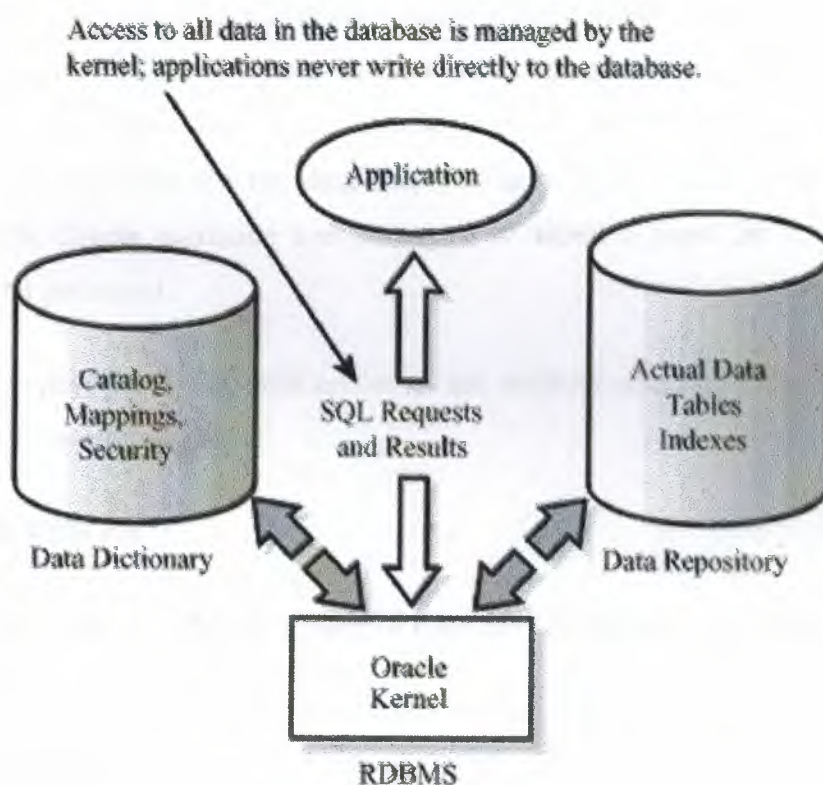


Figure1.2 Structure of Oracle

Not only does the RDBMS take care of locating data, it also determines an optimal access path to store or retrieve the data. Oracle8 uses sophisticated algorithms that enable you to

retrieve information either for the best response for the first set of rows, or for total throughput of all rows to be retrieved.

1.3 ORACLE DATABASE

Physically, an Oracle database is nothing more than a set of files somewhere on disk. The physical location of these files is irrelevant to the function of the database. The files are binary files that we can only access using the Oracle kernel software. Querying data in the database files is typically done with one of the Oracle tools using the Structured Query Language.

Logically, the database is divided into a set of Oracle user accounts, each of which is identified by a username and password unique to that database. Tables and other objects are owned by one of these Oracle users, and access to the data is only available by logging in to the database using an Oracle username and password. Without a valid username and password for the database, you are denied access to anything on the database. The Oracle username and password is different from the operating system username and password.

In addition to physical files, Oracle processes and memory structures must also be present before we can use the database.

1.4 ORACLE FILES

In this part, I discuss the different types of files that Oracle uses on the hard disk drive of any machine.

1.4.1 Database Files

The database files hold the actual data and are typically the largest in size, from a few megabytes to many gigabytes. The other files support the rest of the architecture. Depending on their sizes, the tables and other objects for all the user accounts can obviously go in one database file, but that's not an ideal situation because it does not make the database structure very flexible for controlling access to storage for different

Oracle users, putting the database on different disk drives, or backing up and restoring just part of the database.

We must have at least one database file, but usually, we have many more than one. In terms of accessing and using the data in the tables and other objects, the number or location of the files is immaterial. The database files are fixed in size and never grow bigger than the size at which they were created.

1.4.2 Control Files

Any database must have at least one control file, although we typically have more than one to guard against loss. The control file records the name of the database, the date and time it was created, the location of the database and redo logs, and the synchronization information to ensure that all three sets of files are always in step. Every time we add a new database or redo log file to the database, the information is recorded in the control files.

1.4.3 Redo Logs

Any database must have at least two redo logs. These are the journals for the database, the redo logs record all changes to the user objects or system objects. If any type of failure occurs, such as loss of one or more database files, we can use the changes recorded in the redo logs to bring the database to a consistent state without losing any committed transactions. In the case of non-data loss failure, such as a machine crash, Oracle can apply the information in the redo logs automatically without intervention from the database administrator. The SMON background process automatically reapplies the committed changes in the redo logs to the database files.

Like the other files used by Oracle, the redo log files are fixed in size and never grow dynamically from the size at which they were created.

1.4.3.1 Online Redo Logs

The online redo logs are the two or more redo log files that are always in use while the Oracle instance is up and running. Changes we make are recorded to each of the redo logs in turn. When one is full, the other is written to, when that becomes full, the first is overwritten, and the cycle continues.

1.4.3.2 Offline/Archived Redo Logs

The offline or archived redo logs are exact copies of the online redo logs that have been filled, it is optional whether we ask Oracle to create these. Oracle only creates them when the database is running in ARCHIVELOG mode. If the database is running in ARCHIVELOG mode, the ARCH background process wakes up and copies the online redo log to the offline destination once it becomes full. While this copying is in progress, Oracle uses the other online redo log. If we have a complete set of offline redo logs since the database was last backed up, we have a complete record of changes that have been made. We could then use this record to reapply the changes to the backup copy of the database files if one or more online database files are lost.

1.4.4 Other Supporting Files

When we start an Oracle instance, the instance parameter file determines the sizes and modes of the database. This parameter file is known as the INIT.ORA file. This is an ordinary text file containing parameters for which we can override the default settings. The DBA is responsible for creating and modifying the contents of this parameter file.

On some Oracle platforms, a SGAPAD file is also created, which contains the starting memory address of the Oracle SGA.

1.5 SYSTEM AND USER PROCESSES

In this part, I discuss some of the Oracle system processes that must be running for the database to be useable, including the optional processes and the processes that are created for users connecting to the Oracle database.

1.5.1 Mandatory System Processes

The four Oracle system processes that must always be up and running for the database to be useable include DBWR (Database Writer), LGWR (Log Writer), SMON (System Monitor), and PMON (Process Monitor).

1.5.1.1 DBWR (Database Writer)

The database writer background process writes modified database blocks in the SGA to the database files. It reads only the blocks that have changed. These blocks are also called *dirty* blocks. The database writer writes out the least recently used blocks first. These blocks are not necessarily written to the database when the transaction commits, the only thing that always happens on a commit is that the changes are recorded and written to the online redo log files. The database blocks will be written out later when there are not enough buffers free in the SGA to read in a new block.

1.5.1.2 LGWR (Log Writer)

The log writer process writes the entries in the SGA's redo buffer for one or more transactions to the online redo log files. For example, when a transaction commits, the log writer must write out the entries in the redo log buffer to the redo log files on disk before the process receives a message indicating that the commit was successful. Once committed, the changes are safe on disk even though the modified database blocks are still in the SGA's database buffer area waiting to be written out by DBWR. The SMON can always reapply the changes from the redo logs if the memory's most up-to-date copy of the database blocks is lost.

1.5.1.3 SMON (System Monitor)

The system monitor process looks after the instance. If two transactions are both waiting for each other to release locks and neither of them can continue known as a *deadlock* or *deadly embrace*, SMON detects the situation and one of the processes receives an error message indicating that a deadlock has occurred.

SMON also releases temporary segments that are no longer in use by the user processes which caused them to be created.

During idle periods, SMON compacts the free-space fragments in the database files, making it easier and simpler for Oracle to allocate storage for new database objects or for existing database objects to grow.

In addition, SMON automatically performs recovery when the Oracle instance is first started up, if none of the files have been lost. We won't see a message indicating that instance recovery is occurring, but the instance might take longer to come up.

1.5.1.4 PMON (Process Monitor)

The process monitor monitors the user processes. If any failure occurs with the user processes, PMON automatically rolls back the work of the user process since the transaction started. It releases any locks taken out and other system resources taken up by the failed process.

PMON also monitors the dispatcher and shared server processes, which are part of the multi-threaded server setup, and restarts them if they have died.

1.5.2 Optional System Processes

As well as the four mandatory system processes, there are a number of optional system processes that we can initiate.

1.5.2.1 ARCH (Archiver)

When the database is running in ARCHIVELOG mode and we've started the Archiver background process, it makes a copy of one of the online redo log files to the archive destination. In this way, we can have a complete history of changes made to the database files recorded in the offline and the online redo logs.

There is no point in keeping the Archiver background process running if the database is not running in ARCHIVELOG mode.

1.5.2.2 CKPT (Checkpoint Process)

A checkpoint occurs when one of the online redo log files fills, it will be overwritten when one of the other online redo logs fills. If the redo log file is overwritten, the changes recorded in that file are not available for reapplying in case of system failure. At a checkpoint, the modified database buffer blocks are written down to the relative safety of the database files on disk by the database writer background process. This means that we won't need the record of changes in the event of system failure with lost memory areas. After a checkpoint occurs, the redo log can be reused.

At a checkpoint, all the database file headers and redo log file headers are updated to record the fact that a checkpoint has occurred. The LGWR background process performs the updating task, which could be significant if there are a large number of database and redo log files. The entire database might have to wait for the checkpoint to complete before the redo logs can record further database changes. To reduce the time it takes for LGWR to update the database and redo log file headers, we can initiate the checkpoint process.

A checkpoint can occur at other times, such as when the entries in the redo log files reach a limit defined by the database administrator.

1.5.2.3 RECO (Recoverer)

We use the Recoverer background process when there is a failure in a distributed transaction, and one or more of the databases involved need to either commit or roll back their changes. If initiated, the Recoverer attempts to automatically commit or roll back the transaction on the local database at timed intervals in synchronization with the Recoverer processes on the other Oracle databases.

There is no point in keeping the Recoverer background process running if we're not using distributed transactions on the database.

1.5.2.4 LCK (Lock)

We use the lock background process in the parallel server setup of Oracle where more than one instance is running against the same set of database files. The LCK processes running on all instances will synchronize locking between the instances. If a user connects to one instance and locks a row, the row remains locked for a user attempting to make a change on another instance. Other users can always query the rows regardless of how the rows are locked by other users.

You can initiate up to ten LCK background processes to reduce the bottleneck of synchronizing locking, but one is usually more than enough. You should not initiate the LCK background processes unless you're implementing a parallel server setup of Oracle.

1.5.3 User Processes

User processes logically consist of two halves. The Oracle server code, which translates and executes SQL statements and reads the database files and memory areas, and the tool-specific code, which is the executable code for the tool that is used. The server code is the same regardless of the tool that is executing the SQL statement, the same steps are involved. The server code is sometimes known as the Oracle kernel code.

We can configure the user processes in Oracle three different ways, all of which could coexist for the same instance. These three configurations are single task, dedicated server, or multi-threaded server.

1.5.3.1 Single Task

In the single-task configuration, the tool-specific code and database server code are both configured into one process running on the machine. Each connection to the database has one user process running on the machine.

1.5.3.2 Dedicated Server Processes

In the dedicated server configuration, the two parts of a user process are implemented as two separate processes running on the machine. They communicate with each other using the machine's interprocess communication mechanisms. Each connection to the database has two processes running on the machine. The Oracle kernel software in one process is sometimes called the shadow process.

This configuration is common for UNIX platforms because the operating system cannot protect the Oracle code and memory areas from the application code. It is also common for client/server configurations where the server code resides on the server machine and the tool-specific code runs on the client machine with communication over a network. The way the two component parts of one logical process communicate is fundamentally the same as if one process were implemented on the same machine, except that the two halves of the logical process happen to reside on two machines and communicate over the network using Net8 rather than the interprocess communication mechanisms of the operating system.

The dedicated server configuration can be wasteful because memory is allocated to the shadow process and the number of processes that must be serviced on the machine increases, even when the user is not making any database requests. The dedicated server will only process requests from one associated client process.

1.5.3.3 The Multi-Threaded Server

The multi-threaded server configuration enables one Oracle server process to perform work for many user processes. This overcomes the drawbacks of the dedicated server configuration. It reduces the number of processes running and the amount of memory used on the machine and can improve system performance. The multi-threaded server introduces two new types of system processes that support this part of the architecture.

Using one of the shared server processes that comes as part of the multi-threaded server configuration is not appropriate when a user process is making many database requests

such as an export backup of the database. For that process, we could use a dedicated server. A mixture of both configurations can coexist.

1.6 ORACLE MEMORY

In this part, I discuss how Oracle uses the machine's memory. Generally, the greater the real memory available to Oracle, the quicker the system runs.

1.6.1 System Global Area (SGA)

The system global area, sometimes known as the shared global area, is for data and control structures in memory that can be shared by all the Oracle background and user processes running on that instance. Each Oracle instance has its own SGA. In fact, the SGA and background processes is what defines an instance. The SGA memory area is allocated when the instance is started, and it's flushed and deallocated when the instance is shut down.

The contents of the SGA are divided into three main areas, the database buffer cache, the shared pool area, and the redo cache. The size of each of these areas is controlled by parameters in the INIT.ORA file. The bigger you can make the SGA and the more of it that can fit into the machine's real memory as opposed to virtual memory, the quicker your instance will run.

1.6.1.1 Database Buffer Cache

The database buffer cache of the SGA holds Oracle blocks that have been read in from the database files. When one process reads the blocks for a table into memory, all the processes for that instance can access those blocks.

If a process needs to access some data, Oracle checks to see if the block is already in this cache. If the Oracle block is not in the buffer, it must be read from the database files into the buffer cache. The buffer cache must have a free block available before the data block can be read from the database files.

The Oracle blocks in the database buffer cache in memory are arranged with the most recently used at one end and the least recently used at the other. This list is constantly changing as the database is used. If data must be read from the database files into memory, the blocks at the least recently used end are written back to the database files first. The DBWR process is the only process that writes the blocks from the database buffer cache to the database files. The more database blocks you can hold in real memory, the quicker your instance will run.

1.6.1.2 Redo Cache

The online redo log files record all the changes made to user objects and system objects. Before the changes are written out to the redo logs, Oracle stores them in the redo cache memory area. For example, the entries in the redo log cache are written down to the online redo logs when the cache becomes full or when a transaction issues a commit. The entries for more than one transaction can be included together in the same disk write to the redo log files.

The LGWR background process is the only process that writes out entries from this redo cache to the online redo log files.

1.6.1.3 Shared Pool Area

The shared pool area of the SGA has two main components, the SQL area and the dictionary cache. You can alter the size of these two components only by changing the size of the entire shared pool area.

1.6.1.4 SQL Area

A SQL statement sent for execution to the database server must be parsed before it can execute. The SQL area of the SGA contains the binding information, run-time buffers, parse tree, and execution plan for all the SQL statements sent to the database server. Because the shared pool area is a fixed size, you might not see the entire set of statements

that have been executed since the instance first came up, Oracle might have flushed out some statements to make room for others.

If a user executes a SQL statement, that statement takes up memory in the SQL area. If another user executes exactly the same statement on the same objects, Oracle doesn't need to reparse the second statement because the parse tree and execution plan is already in the SQL area. This part of the architecture saves on reparsing overhead. The SQL area is also used to hold the parsed, compiled form of PL/SQL blocks, which can also be shared between user processes on the same instance.

1.6.1.5 Dictionary Cache

The dictionary cache in the shared pool area holds entries retrieved from the Oracle system tables, otherwise known as the Oracle data dictionary. The data dictionary is a set of tables located in the database files, and because Oracle accesses these files often, it sets aside a separate area of memory to avoid disk I/O.

The cache itself holds a subset of the data from the data dictionary. It is loaded with an initial set of entries when the instance is first started and then populated from the database data dictionary as further information is required. The cache holds information about all the users, the tables and other objects, the structure, security, storage, and so on.

The data dictionary cache grows to occupy a larger proportion of memory within the shared pool area as needed, but the size of the shared pool area remains fixed.

1.6.2 Process Global Area

The process global area, sometimes called the program global area or PGA, contains data and control structures for one user or server process. There is one PGA for each user process to the database.

The actual contents of the PGA depend on whether the multi-threaded server configuration is implemented, but it typically contains memory to hold the session's variables, arrays, some rows results, and other information. If you're using the multi-

threaded server, some of the information that is usually held in the PGA is instead held in the common SGA.

The size of the PGA depends on the operating system used to run the Oracle instance, and once allocated, it remains the same. Memory used in the PGA does not increase according to the amount of processing performed in the user process. The database administrator can control the size of the PGA by modifying some of the parameters in the instance parameter file INIT.ORA.

CHAPTER 2

2.1 ORACLE ACCESS WITH JDBC

Java is designed to be platform independent. A pure Java program written for a Windows machine will run without recompilation on a Solaris Sparc, an Apple Macintosh, or any platform with the appropriate Java virtual machine.

JDBC extends this to databases. If we write a Java program with JDBC, given the appropriate database driver, that program will run against any database without having to recompile the Java code. Without JDBC, our Java code would need to run platform specific native database code, thus violating the Java motto, Write Once, Run Anywhere. JDBC allows us to write Java code, and leave the platform specific code to the driver. In the event we change databases, we simply change the driver used by our Java code and we are immediately ready to run against the new database.

JDBC is a rich set of classes that give us transparent access to a database with a single application programming interface, or API. This access is done with plug-in platform-specific modules, or drivers. Using these drivers and the JDBC classes, our programs will be able to access consistently any database that supports JDBC, giving us total freedom to concentrate on our applications and not to worry about the underlying database.

All access to JDBC data sources is done through SQL. Sun has concentrated on JDBC issuing SQL commands and retrieving their results in a consistent manner. Though we gain so much ease by using this SQL interface, we do not have the raw database access that we might be used to. With the classes we can open a connection to a database, execute SQL statements, and do what we will with the results.

2.1.1 Driver Types

As mentioned above, our Java JDBC code is portable because the database specific code is contained in a Java class known as the driver. The two most common kinds of driver for connecting to an Oracle database are the thin driver and the OCI driver.

2.1.1.1 Thin driver

The thin driver is known as a Type IV driver, it is a pure Java driver that connects to a database using the database's native protocol. While we can use the thin driver in any environment, the Type IV driver is intended for use in Java applets and other client-side programs. A Java client can be run on any platform. For that reason, the JDBC driver downloaded with an applet or used by a Java client may not have access to platform native code and must be pure Java.

2.1.1.2 OCI8 driver

The OCI8 driver is known as a Type II driver. It uses platform native code to call the database. Because it uses a native API, it can connect to and access a database faster than the thin driver. For the same reason, the Type II driver cannot be used where the program does not have access to the native API. This usually applies to applets and other client programs which may be deployed on any arbitrary platform.

2.1.2 The DriverManager Class

The cornerstone of the JDBC package is the DriverManager class. This class keeps track of all the different available database drivers. We won't usually see the DriverManager's work, though. This class mostly works behind the scenes to ensure that everything is cool for our connections.

The DriverManager maintains a Vector that holds information about all the drivers that it knows about. The elements in the Vector contain information about the driver such as the class name of the Driver object, a copy of the actual Driver object, and the Driver security context.

The DriverManager, while not a static class, maintains all static instance variables with static access methods for registering and unregistering drivers. This allows the DriverManager never to need instantiation. Its data always exists as part of the Java runtime. The drivers managed by the DriverManager class are represented by the Driver class.

2.1.3 The Driver Class

If the cornerstone of JDBC is the DriverManager, then the Driver class is most certainly the bricks that build the JDBC. The Driver is the software wedge that communicates with the platform-dependent database, either directly or using another piece of software. How it communicates really depends on the database, the platform, and the implementation.

It is the Driver's responsibility to register with the DriverManager and connect with the database. Database connections are represented by the Connection class.

2.1.4 The Connection Class

The Connection class encapsulates the actual database connection into an easy-to-use package. Sticking with our foundation building analogy here, the Connection class is the mortar that binds the JDBC together. It is created by the DriverManager when its getConnection() method is called. This method accepts a database connection URL and returns a database Connection to the caller.

When we call the getConnection() method, the DriverManager asks each driver that has registered with it whether the database connection URL is valid. If one driver responds

positively, the DriverManager assumes a match. If no driver responds positively, an SQLException is thrown. The DriverManager returns the error "no suitable driver," which means that of all the drivers that the DriverManager knows about, not one of them could figure out the URL you passed to it.

Assuming that the URL was good and a Driver loaded, then the DriverManager will return a Connection object to us. What can we do with a Connection object? Not much. This class is nothing more than an encapsulation of our database connection. It is a factory and manager object, and is responsible for creating and managing Statement objects.

2.1.5 The Statement Class

Picture the Connection as an open pipeline to our database. Database transactions travel back and forth between our program and the database through this pipeline. The Statement class represents these transactions.

The Statement class encapsulates SQL queries to our database. Using several methods, these calls return objects that contain the results of our SQL query. When we execute an SQL query, the data that is returned to us is commonly called the result set.

2.1.6 The ResultSet Class

As we've probably guessed, the ResultSet class encapsulates the results returned from an SQL query. Normally, those results are in the form of rows of data. Each row contains one or more columns. The ResultSet class acts as a cursor, pointing to one record at a time, enabling us to pick out the data we need.

2.2 SQLJ

SQLJ enables us to embed static SQL operations in Java code in a way that is compatible with the Java design philosophy. A SQLJ program is a Java program containing embedded static SQL statements that comply with the ANSI-standard SQLJ Language Reference syntax. Static SQL operations are predefined, the operations themselves do not change in real-time as a user runs the application, although the data values that are transmitted can change dynamically. Typical applications contain much more static SQL than dynamic SQL. Dynamic SQL operations are *not* predefined, the operations themselves can change in real-time and require direct use of JDBC statements. However, we can use SQLJ statements and JDBC statements in the same program.

SQLJ consists of both a translator and a runtime component and is smoothly integrated into our development environment. The developer runs the translator, with translation, compilation, and customization taking place in a single step when the sqlj front-end utility is run. The translation process replaces embedded SQL with calls to the SQLJ runtime, which implements the SQL operations. In standard SQLJ this is typically, but not necessarily, performed through calls to a JDBC driver. In the case of an Oracle database, we would typically use an Oracle JDBC driver. When the end user runs the SQLJ application, the runtime is invoked to handle the SQL operations.

The Oracle SQLJ translator is conceptually similar to other Oracle precompilers and allows the developer to check SQL syntax, verify SQL operations against what is available in the schema, and check the compatibility of Java types with corresponding database types. In this way, errors can be caught by the developer instead of by a user at runtime.

The SQLJ methodology of embedding SQL operations directly in Java code is much more convenient and concise than the JDBC methodology. In this way, SQLJ reduces development and maintenance costs in Java programs that require database connectivity. When dynamic SQL is required, however, SQLJ supports interoperability with JDBC

such that we can intermix SQLJ code and JDBC code in the same source file. Alternatively, we can use PL/SQL blocks within SQLJ statements for dynamic SQL.

2.2.1 SQLJ Components

Oracle SQLJ consists of two major components.

2.2.1.1 Oracle SQLJ Translator

This component is a precompiler that developers run after creating SQLJ source code. The translator, written in pure Java, supports a programming syntax that allows us to embed SQL operations inside SQLJ executable statements. SQLJ executable statements, as well as SQLJ declarations, are preceded by the `#sql` token and can be interspersed with Java statements in a SQLJ source code file. SQLJ source code file names must have the `.sqlj` extension.

The translator produces a `.java` file and one or more SQLJ profiles, which contain information about our SQL operations. SQLJ then automatically invokes a Java compiler to produce `.class` files from the `.java` file.

2.2.1.2 Oracle SQLJ Runtime

This component is invoked automatically each time an end user runs a SQLJ application. The SQLJ runtime, also written in pure Java, implements the desired actions of our SQL operations, accessing the database using a JDBC driver. The generic SQLJ standard does not require that a SQLJ runtime use a JDBC driver to access the database, however, the Oracle SQLJ runtime does require a JDBC driver, and, in fact, requires an Oracle JDBC driver if our application is customized with the default Oracle customizer.

In addition to the translator and runtime, there is a component known as the customizer. A customizer tailors our SQLJ profiles for a particular database implementation and vendor-specific features and datatypes. By default, the Oracle SQLJ front end invokes an Oracle customizer to tailor our profiles for an Oracle database and Oracle-specific features and datatypes.

When we use the Oracle customizer during translation, our application will require the Oracle SQLJ runtime and an Oracle JDBC driver when it runs.

2.2.1.3 SQLJ Profiles

SQLJ profiles are serialized Java resources generated by the SQLJ translator, which contain details about the embedded SQL operations in our SQLJ source code. The translator creates these profiles, then either serializes them and puts them into binary resource files, or puts them into .class files according to our translator option settings.

SQLJ profiles are used in implementing the embedded SQL operations in our SQLJ executable statements. Profiles contain information about our SQL operations and the types and modes of data being accessed. A profile consists of a collection of entries, where each entry maps to one SQL operation. Each entry fully specifies the corresponding SQL operation, describing each of the parameters used in executing this instruction.

SQLJ generates a profile for each connection context class in our application, where, typically, each connection context class corresponds to a particular set of SQL entities we use in our database operations. The SQLJ standard requires that the profiles be of standard format and content. Therefore, for our application to use vendor-specific extended features, our profiles must be customized. By default, this occurs automatically, with our profiles being customized to use Oracle-specific extended features.

2.2.2 Oracle Extensions to the SQLJ Standard

Beginning with Oracle8i, Oracle SQLJ supports the SQLJ ISO specification. Because the SQLJ ISO standard is a superset of the SQLJ ANSI standard, it requires a JDK 1.2 or later environment that complies with J2EE. The SQLJ ANSI standard requires only JDK 1.1.x. The Oracle SQLJ translator accepts a broader range of SQL syntax than the ANSI SQLJ Standard specifies.

The ANSI standard addresses only the SQL92 dialect of SQL, but allows extension beyond that. Oracle SQLJ supports Oracle's SQL dialect, which is a superset of SQL92. If we need to create SQLJ programs that work with other DBMS vendors, avoid using SQL syntax and SQL types that are not in the standard and, therefore, may not be supported in other environments.

2.2.3 Basic Translation Steps and Runtime Processing

2.2.3.1 Translation Steps

The following sequence of events occurs, presuming each step completes without fatal error.

1. The JVM invokes the SQLJ translator.
2. The translator parses the source code in the .sqlj file, checking for proper SQLJ syntax and looking for type mismatches between our declared SQL datatypes and corresponding Java host variables.
3. The translator invokes the semantics-checker, which checks the semantics of embedded SQL statements.

The developer can use online or offline checking, according to SQLJ option settings. If online checking is performed, then SQLJ will connect to the database to verify that the database supports all the database tables, stored procedures, and SQL syntax that the application uses, and that the host variable types in the SQLJ application are compatible with datatypes of corresponding database columns.

4. The translator processes our SQLJ source code, converts SQL operations to SQLJ runtime calls, and generates Java output code and one or more SQLJ profiles. A separate profile is generated for each connection context class in our source code, where a different connection context class is typically used for each interrelated set of SQL entities that we use in our database operations.

5. The JVM invokes the Java compiler, which is usually, but not necessarily, the standard javac provided with the Sun Microsystems JDK.
6. The compiler compiles the Java source file generated in step 4 and produces Java .class files as appropriate. This will include a .class file for each class we defined, a .class file for each of our SQLJ declarations, and a .class file for the profile-keys class.
7. The JVM invokes the Oracle SQLJ customizer or other specified customizer.
8. The customizer customizes the profiles generated in step 4.

2.2.3.2 Runtime Processing

When a user runs the application, the SQLJ runtime reads the profiles and creates "connected profiles", which incorporate database connections. Then the following occurs each time the application must access the database.

1. SQLJ-generated application code uses methods in a SQLJ-generated profile-keys class to access the connected profile and read the relevant SQL operations. There is mapping between SQLJ executable statements in the application and SQL operations in the profile.
2. The SQLJ-generated application code calls the SQLJ runtime, which reads the SQL operations from the profile.
3. The SQLJ runtime calls the JDBC driver and passes the SQL operations to the driver.
4. The SQLJ runtime passes any input parameters to the JDBC driver.
5. The JDBC driver executes the SQL operations.
6. If any data is to be returned, the database sends it to the JDBC driver, which sends it to the SQLJ runtime for use by our application.

2.2.4 SQLJ Declarations

A SQLJ declaration consists of the `#sql` token followed by the declaration of a class. SQLJ declarations introduce specialized Java types into our application. There are currently two kinds of SQLJ declarations, iterator declarations and connection context declarations, defining Java classes.

Iterator declarations define iterator classes. Iterators are conceptually similar to JDBC result sets and are used to receive multi-row query data. An iterator is implemented as an instance of an iterator class.

Connection context declarations define connection context classes. Each connection context class is typically used for connections whose operations use a particular set of SQL entities. That is to say, instances of a particular connection context class are used to connect to schemas that include SQL entities with the same names and characteristics. SQLJ implements each database connection as an instance of a connection context class.

2.2.5 Java Host Expressions, Context Expressions, and Result Expressions

There are three categories of Java expressions used in SQLJ code: host expressions, context expressions, and result expressions. Host expressions are the most frequently used and merit the most discussion.

SQLJ uses Java host expressions to pass arguments between your Java code and your SQL operations. This is how you pass information between Java and SQL. Host expressions are interspersed within the embedded SQL operations in SQLJ source code.

The most basic kind of host expression, consisting of only a Java identifier, is referred to as a host variable. A context expression specifies a connection context instance or execution context instance to be used for a SQLJ statement. A result expression specifies an output variable for query results or a function return.

2.2.5.1 Host Expressions

Any valid Java expression can be used as a host expression. In the simplest case, which is typical, the expression consists of just a single Java variable. Other kinds of host expressions include: arithmetic expressions, Java method calls with return values, Java class field values, array elements, conditional expressions, logical expressions, or bitwise expressions.

Java identifiers used as host variables or in host expressions can represent any of the following:

1. Local variables.
2. Declared parameters.
3. Class fields.
4. Static or instance method calls

Local variables used in host expressions can be declared anywhere that other Java variables can be declared. Fields can be inherited from a superclass.

Java variables that are legal in the Java scope where the SQLJ executable statement appears can be used in a host expression in a SQL statement, presuming its type is convertible to or from a SQL datatype. Host expressions can be input, output, or input-output.

2.2.5.2 Context Expressions

A context expression is an input expression that specifies the name of a connection context instance or an execution context instance to be used in a SQLJ executable statement. Any legal Java expression that yields such a name can be used.

2.2.5.3 Result Expressions

A result expression is an output expression used for query results or a function return. It can be any legal Java expression that is *assignable*, meaning that it can logically appear on the left side of an equals sign.

Result expressions and context expressions appear lexically in the SQLJ space, unlike host expressions, which appear lexically in the SQL space. Therefore, a result expression or context expression must *not* be preceded by a colon.

2.2.6 Stored Procedure and Function Calls

SQLJ provides convenient syntax for calling stored procedures and stored functions in the database. These procedures and functions could be written in Java, PL/SQL, or any other language supported by the database.

A stored function requires a result expression in our SQLJ executable statement to accept the return value and can optionally take input, output, or input-output parameters as well.

A stored procedure does not have a return value but can optionally take input, output, or input-output parameters. A stored procedure can return output through any output or input-output parameter.

2.2.7 Multithreading in SQLJ

We can use SQLJ in writing multithreaded applications; however, any use of multithreading in our SQLJ application is subject to the limitations of our JDBC driver. This includes any synchronization limitations.

We are required to use a different execution context instance for each thread. We can accomplish this in one of two ways.

1. Specify connection context instances for our SQLJ statements such that a different connection context instance is used for each thread. Each connection context instance automatically has its own default execution context instance.
2. If we are using the same connection context instance with multiple threads, then declare additional execution context instances and specify execution context instances for our SQLJ statements such that a different execution context instance is used for each thread.

If we are using one of the Oracle JDBC drivers, multiple threads can use the same connection context instance as long as different execution context instances are specified and there are no synchronization requirements directly visible to the user. However, that database access is sequential, only one thread is accessing the database at any given time.

If a thread attempts to execute a SQL operation that uses an execution context that is in use by another operation, then the thread is blocked until the current operation completes. If an execution context were shared between threads, the results of a SQL operation performed by one thread would be visible in the other thread. If both threads were executing SQL operations, a race condition might occur, the results of an execution in one thread might be overwritten by the results of an execution in the other thread before the first thread had processed the original results. This is why multiple threads are not allowed to share an execution context instance.

2.2.8 SQLJ and JDBC Interoperability

We can use SQLJ statements for static SQL operations, but not for dynamic operations. We can, however, use JDBC statements for dynamic SQL operations, and there might be situations where our application will require both static and dynamic SQL operations. SQLJ allows us to use SQLJ statements and JDBC statements concurrently and provides interoperability between SQLJ constructs and JDBC constructs.

Two kinds of interactions between SQLJ and JDBC are particularly useful:

- between SQLJ connection contexts and JDBC connections

- between SQLJ iterators and JDBC result sets

2.2.8.1 Converting from Connection Contexts to JDBC Connections

If we want to perform a dynamic SQL operation through a database connection that we have established in SQLJ, then we must convert the SQLJ connection context instance to a JDBC connection instance.

Any connection context instance in a SQLJ application, whether an instance of the `sqlj.runtime.ref.DefaultContext` class or of a declared connection context class, contains an underlying JDBC connection instance and a `getConnection()` method that returns that JDBC connection instance. Use the JDBC connection instance to create JDBC statement objects if you want to use any dynamic SQL operations.

2.2.8.2 Converting from JDBC Connections to Connection Contexts

If we initiate a connection as a JDBC Connection or `OracleConnection` instance but later want to use it as a SQLJ connection context instance, then we can convert the JDBC connection instance to a SQLJ connection context instance.

The `DefaultContext` class and all declared connection context classes have a constructor that takes a JDBC connection instance as input and constructs a SQLJ connection context instance.

2.2.8.3 Shared Connections

A SQLJ connection context instance and the associated JDBC connection instance share the same underlying database connection. When we get a JDBC connection instance from a SQLJ connection context instance, the Connection instance inherits the state of the connection context instance. Among other things, the Connection instance will retain the auto-commit setting of the connection context instance.

When we construct a SQLJ connection context instance from a JDBC connection instance, the connection context instance inherits the state of the Connection instance.

Among other things, the connection context instance will retain the auto-commit setting of the Connection instance.

Given a SQLJ connection context instance and associated JDBC connection instance, calls to methods that alter session state in one instance will also affect the other instance, because it is actually the underlying shared database session that is being altered.

Because there is just a single underlying database connection, there is also a single underlying set of transactions. A COMMIT or ROLLBACK operation in one connection instance will affect any other connection instances that share the same underlying connection.

2.2.9 SQLJ In the Server

SQLJ code, as with any Java code, can run in the Oracle8i server in stored procedures, stored functions, triggers, Enterprise JavaBeans, or CORBA objects. Database access is through a server-side implementation of the SQLJ runtime in combination with the Oracle JDBC server-side internal driver.

In addition, an embedded SQLJ translator in the Oracle8i server is available to translate SQLJ source files directly in the server.

Considerations for running SQLJ in the server include several server-side coding issues as well as decisions about where to translate our code and how to load it into the server. We must also be aware of how the server determines the names of generated output. We can either translate and compile on a client and load the class and resource files into the server, or we can load .sqlj source files into the server and have the files automatically translated by the embedded SQLJ translator.

The embedded translator has a different user interface than the client-side translator. Supported options can be specified using a database table, and error output is to a database table. Output files from the translator, .java and .ser, are transparent to the developer.

2.2.9.1 Creating SQLJ Code for Use within the Server

With few exceptions, writing SQLJ code for use within the target Oracle8i server is identical to writing SQLJ code for client-side use. The few differences are due to Oracle JDBC characteristics or general Java characteristics in the server, rather than being specific to SQLJ.

2.2.9.2 Database Connections within the Server

The concept of connecting to a server is different when our SQLJ code is running within this server itself, there is no explicit database connection. By default, an implicit channel to the database is employed for any Java program running in the server. We do not have to initialize this connection, it is automatically initialized for SQLJ programs. We do not have to register or specify a driver, create a connection instance, specify a default connection context, specify any connection objects for any of our #sql statements, or close the connection.

2.2.9.3 Coding Issues within the Server

Result sets issued by the internal driver persist across calls, and their finalizers do not release their database cursors. Because of this, it is especially important to close all iterators to avoid running out of available cursors, unless we have a particular reason for keeping an iterator open.

The internal driver does not support auto-commit functionality, the auto-commit setting is ignored within the server. Use explicit COMMIT or ROLLBACK statements to implement or cancel your database updates.

2.2.9.4 Name Resolution in the Server

Class loading and name resolution in the server follow a very different paradigm than on a client, because the environments themselves are very different. Java name resolution in the Oracle8i JVM includes the following:

1. Class resolver specs, which are schema lists to search in resolving a class schema object.
2. The resolver, which maintains mappings between class schema objects that reference each other in the server.

A class schema object is said to be resolved when all of its external references to Java names are bound. In general, all the classes of a Java program should be compiled or loaded before they can be resolved.

When all the class schema objects of a Java program in the server are resolved and none of them have been modified since being resolved, the program is effectively pre-linked and ready to run.

A class schema object must be resolved before Java objects of the class can be instantiated or methods of the class can be executed.

2.2.9.5 SQL Names versus Java Names

SQL names such as names of source, class, and resource schema objects are not global in the way that Java names are global. The Java Language Specification directs that package names use Internet naming conventions to create globally unique names for Java programs. By contrast, a fully qualified SQL name is interpreted only with respect to the current schema and database.

Because of this inherent difference, SQL names must be interpreted and processed differently from Java names. SQL names are relative names and are interpreted from the point of view of the schema where a program is executed. This is central to how the program binds local data stored at that schema. Java names are global names, and the classes that they designate can be loaded at any execution site, with reasonable expectation that those classes will be classes that were used to compile the program.

2.2.9.6 Translating SQLJ Source on a Client and Loading Components

One approach to developing SQLJ code for the server is to first run the SQLJ translator on a client machine to take care of translation, compilation, and profile customization. Then load the resulting class and resource files including SQLJ profiles into the server, typically using a Java archive file.

If we are developing our source on a client machine, and have a SQLJ translator available there, this approach is advisable. It allows the most flexibility in running the translator, because option-setting and error-processing are not as convenient in the server.

It might also be advisable to use the SQLJ `-ser2class` option during translation when you intend to load an application into the server. This results in SQLJ profiles being converted from `.ser` serialized resource files to `.class` files and simplifies their naming. However, profiles converted to `.class` files cannot be further customized. To further customize, we would have to rerun the translator and regenerate the profiles.

When we load `.class` files and `.ser` resource files into the server, either directly or using a `.jar` file, the resulting database library units are referred to as Java class schema objects, for Java classes and Java resource schema objects, for Java resources. Our SQLJ profiles will be in resource schema objects if we load them as `.ser` files, or in class schema objects if we enabled `-ser2class` during translation and load them as `.class` files.

2.2.9.7 Error Output from the Server Embedded Translator

SQLJ error processing in the server is similar to general Java error processing in the server. SQLJ errors are directed into the `USER_ERRORS` table of the user schema. We can `SELECT` from the `TEXT` column of this table to get the text of a given error message.

Informational messages and suppressable warnings are withheld by the server-side translator in a way that is equivalent to the operation of the client-side translator.

2.3.1 Introduction to Net8

Net8 enables the machines in our network to communicate with one another. It facilitates and manages communication sessions between a client application and a remote database. Specifically, Net8 performs three basic operations.

1. **Connection:** opening and closing connections between a client or a server acting as a client and a database server over a network protocol.
2. **Data Transport:** packaging and sending data such as SQL statements and data responses so that it can be transmitted and understood between a client and a server.
3. **Exception Handling:** initiating interrupt requests from the client or server.

2.3.2 Advantages of Net8

Net8 provides the following benefits to users of networked applications.

2.3.2.1 Network Transparency

Net8 provides support for a broad range of network transport protocols including TCP/IP, SPX/IPX, IBM LU6.2, Novell, and DECnet. It does so in a manner that is invisible to the application user. This enables Net8 to interoperate across different types of computers, operating systems, and networks to transparently connect any combination of PC, UNIX, legacy, and other system without changes to the existing infrastructure.

2.3.2.2 Protocol Independence

Net8 enables Oracle applications to run over any supported network protocol by using the appropriate Oracle Protocol Adapter. Applications can be moved to another protocol stack by installing the necessary Oracle Protocol Adapter and the industry protocol stack. Oracle Protocol Adapters provide Net8 access to connections over specific protocols or networks. On some platforms, a single Oracle Protocol Adapter will operate on several different network interface boards, allowing you to deploy applications in any networking environment.

2.3.2.3 Media/Topology Independence

When Net8 passes control of a connection to the underlying protocol, it inherits all media and/or topologies supported by that network protocol stack. This allows the network protocol to use any means of data transmission, such as Ethernet, Token Ring, or other, to accomplish low level data link transmissions between two machines.

2.3.2.4 Heterogeneous Networking

Oracle's client-server and server-server models provide connectivity between multiple network protocols using Oracle Connection Manager.

2.3.2.5 Large Scale Scalability

By enabling us to use advanced connection concentration and connection pooling features, Net8 makes it possible for thousands of concurrent users to connect to a server.

2.3.3 Net8 Features

Net8 Release 8.0 features several enhancements that extend scalability, manageability and security for the Oracle network.

2.3.3.1 Scalability Features

Scalability refers to the ability to support simultaneous network access by a large number of clients to a single server. With Net8, this is accomplished by optimizing the usage of network resources by reducing the number of physical network connections a server must maintain. Net8 offers improved scalability through two new features.

1. Connection pooling.
2. Connection concentration.

Both of these features optimize usage of server network resources to eliminate data access bottlenecks and enable large numbers of concurrent clients to access a single

server. Additionally, other enhancements such as a new buffering methods and asynchronous operations further improve Net8 performance.

2.3.3.2 Manageability Features

Net8 introduces a number of new features that will simplify configuration and administration of the Oracle network for both workgroup and enterprise environments.

For workgroup environments, Net8 offers simple configuration-free connectivity through installation defaults and a new name resolution feature called host naming. For enterprise environments, Net8 centralizes client administration and simplifies network management with Oracle Names. In addition to these new features, Net8 introduces the Oracle Net8 Assistant.

2.3.3.2.1 Host Naming

Host Naming refers to a new naming method which resolves service names to network addresses by enlisting the services of existing TCP/IP hostname resolution systems. Host Naming can eliminate the need for a local naming configuration file in environments where simple database connectivity is desired.

2.3.3.2.2 Oracle Net8 Assistant

The Oracle Net8 Assistant is a new end user, stand-alone Java application that can be launched either as a stand-alone application or from the Oracle Enterprise Manager console. It automates client configuration and provides an easy-to-use interface as well as wizards to configure and manage Net8 networks.

Because the Oracle Net8 Assistant is implemented in Java, it is available on any platform that supports the Java Virtual Machine.

2.3.3.3 Multiprotocol Support Using Oracle Connection Manager

Oracle Connection Manager provides the capability to seamlessly connect two or more network protocol communities, enabling transparent Net8 access across multiple protocols. In this sense, it replaces the functionality provided by the Oracle MultiProtocol Interchange with SQL*Net. Oracle Connection Manager can also be used to provide network access control. For example, links processed through Oracle Connection Manager can be filtered on the basis of origin, destination, or user ID. It incorporates a Net8 application proxy for implementing firewall-like functionality.

2.3.3.4 Oracle Trace Assistant

Net8 includes the Oracle Trace Assistant to help decode and analyze the data stored in Net8 trace files. The Oracle Trace Assistant provides an easy way to understand and take advantage of the information stored in trace files, it is useful for diagnosing network problems and analyzing network performance. It can be used to better pinpoint the source of a network problem or identify a potential performance bottleneck.

2.3.3.5 Native Naming Adapters

Native Naming Adapters, previously bundled with the Advanced Networking Option, are now included with Net8. These adapters provide native support for industry-standard name services, including Sun NIS/Yellow Pages and Novell NetWare Directory Services (NDS).

2.3.4 Net8 Operations

Net8 is responsible for enabling communications between the cooperating partners in an Oracle distributed transaction, whether they be client-server or server-server. Specifically, Net8 provides three basic networking operations:

1. Connect Operations.
2. Data Operations.
3. Exception Operations.

2.3.5 Connect Operations

Net8 supports two types of connect operations.

2.3.5.1 Connecting to Servers

Users initiate a connect request by passing information such as a username and password along with a short name for the database service that they wish to connect. That short name, called a *service name*, is mapped to a network address contained in a *connect descriptor*. Depending upon our specific network configuration, this connect descriptor may be stored in one of the following.

1. A local names configuration file called TNSNAMES.ORA.
2. A Names Server for use by Oracle Names.
3. A native naming service such as NIS or DCE CDS.

Net8 coordinates its sessions with the help of a network listener.

2.3.5.2 Establishing Connections with the Network Listener

The network listener is a single process or task setup specifically to receive connection requests on behalf of an application. Listeners are configured to "listen on" an address specified in a listener configuration file for a database or non-database service. Once started, the listener will receive client connect requests on behalf of a service, and respond in one of three ways:

1. Bequeath the session to a new dedicated server process.
2. Redirect to an existing server process.
3. Refuse the session.

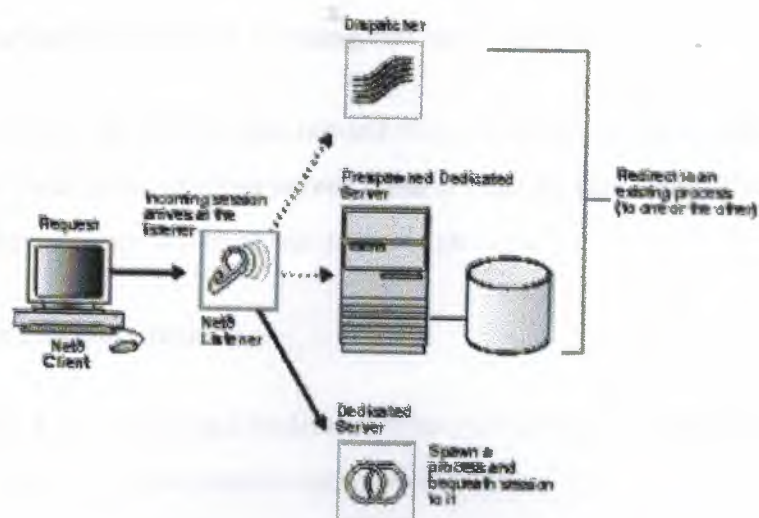


Figure2.1 Network Listener In a Typical Net8 Connection

2.3.5.2.1 Bequeathed Sessions to Dedicated Server Processes

If the listener and server exist on the same node, the listener may create or spawn dedicated server processes as connect requests are received. Dedicated server processes are committed to one session only and exist for the duration of that session.

When a client disconnects, the dedicated server process associated with the client closes. Figure 2.2 depicts the role of the network listener in a bequeathed connection to a dedicated server process.

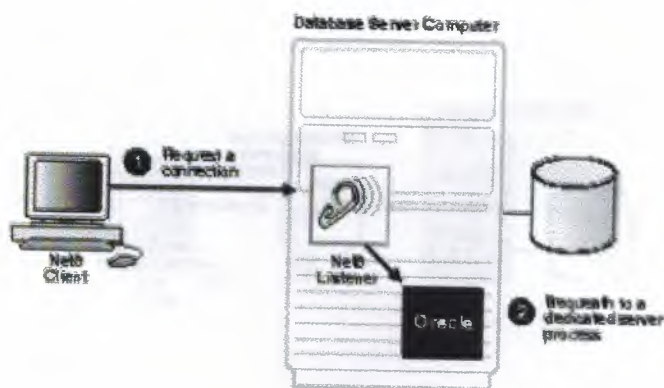


Figure2.2 Bequeathed Connection To a Dedicated Server Process

2.3.5.2.2 Redirected Sessions to Existing Server Processes

Alternatively, Net8 may redirect the request to an existing server process. It does this by sending the address of an existing server process back to the client. The client will then resend its connect request to the server address provided.

Existing server processes include.

1. Prestarted or Prespawnd Dedicated Server Processes by the listener.
2. Dispatcher Processes created outside the listener process.

1. Prespawnd Dedicated Server Processes

Net8 provides the option of automatically creating dedicated server processes before the request is received. These processes last for the life of the listener, and can be reused by subsequent connection requests. The use of prespawnd dedicated server processes requires specification in a listener configuration file.

When clients disconnect, the prespawnd dedicated server process associated with the client returns to the idle pool. It then waits a specified length of time to be assigned to another client. If no client is handed to the prespawnd server before the timeout expires, the prespawnd server shuts down. Figure 2-3 depicts the role of the network listener in a redirected connection to a prespawnd dedicated server process.

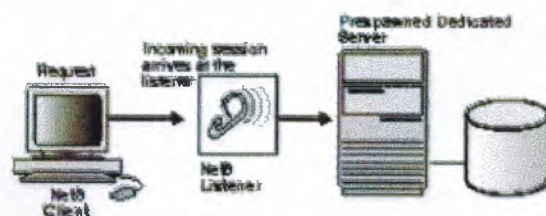


Figure2.3 Redirected Connection To a Prespawnd Dedicated Server Process

2. Dispatcher Server Processes

A dispatcher server process enables many clients to connect to the same server without the need for a dedicated server process for each client. It does this with the help of a dispatcher which handles and directs multiple incoming session requests to the shared server.

When an Oracle server has been configured as a multi-threaded server, incoming sessions are always routed to the dispatcher unless either the session specifically requests a dedicated server or no dispatchers are available.

Once the dispatcher addresses are registered, the listener can redirect incoming connect requests to them. The listener and the Oracle dispatcher server are now ready to receive incoming sessions.

When clients disconnect, the shared server associated with the client stays active and processes other incoming requests. Figure 2.4 depicts the role of the network listener in a redirected connection to a dispatcher server process.

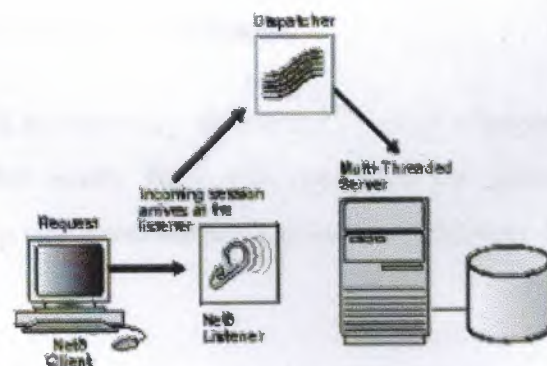


Figure 2.4 Redirected Connection To a Dispatcher Server Process

2.3.5.2.3 Refused Sessions

The network listener will refuse a session in the event that it does not know about the server being requested, or if the server is unavailable. It refuses the session by generating and sending a refuse response packet back to the client.

2.3.6 Disconnecting from Servers

Requests to disconnect from the server can be initiated in the following ways.

2.2.6.1 User-Initiated Disconnect

A user can request a disconnection from the server when a client-server transaction completes. A server can also disconnect from a second server when all server-server data transfers have been completed, and no need for the link remains.

2.3.6.2 Additional Connection Request

If a client application is connected to a server and requires access to another user account on the same or other server, most Oracle tools will first disconnect the application from the server to which it is currently connected. Once the disconnection is completed, a connection request to the new user account on the appropriate server is initiated.

2.3.6.3 Abnormal Connection Termination

Other components will occasionally disconnect or abort communications without giving notice to Net8. In this event, Net8 will recognize the failure during its next data operation, and clean up client and server operations, effectively disconnecting the current operation.

2.3.6.4 Timer Initiated Disconnect or Dead Connection Detection

Dead connection detection is a feature that allows Net8 to identify connections that have been left hanging by the abnormal termination of a client. On a connection with dead

connection detection enabled, a small probe packet is sent from server to client at a user-defined interval. If the connection is invalid, the connection will be closed when an error is generated by the send operation, and the server process will terminate the connection.

This feature minimizes the waste of resources by connections that are no longer valid. It also automatically forces a database rollback of uncommitted transactions and locks held by the user of the broken connection.

2.3.7 Data Operations

Net8 supports four sets of client-server data operations.

1. Send data synchronously.
2. Receive data synchronously.
3. Send data asynchronously.
4. Receive data asynchronously.

On the client side, a SQL dialogue request is forwarded using a send request in Net8. On the server side, Net8 processes a receive request and passes the data to the database. The opposite occurs in the return trip from the server.

Basic send and receive requests are synchronous. When a client initiates a request, it waits for the server to respond with the answer. It can then issue an additional request.

Net8 adds the capability to send and receive data requests asynchronously. This capability was added to support the Oracle shared server, also called a multi-threaded server, which requires asynchronous calls to service incoming requests from multiple clients.

2.3.8 Exception Operations

Net8 supports three types of exception operations.

1. Initiate a break over the connection.

2. Reset a connection for synchronization after a break.
3. Test the condition of the connection for incoming break.

The user controls only one of these three operations, that is, the initiation of a break. When the user presses the Interrupt key, the application calls this function. Additionally, the database can initiate a break to the client if an abnormal operation occurs, such as during an attempt to load a row of invalid data using SQL*Loader.

The other two exception operations are internal to products that use Net8 to resolve network timing issues. Net8 can initiate a test of the communication channel, for example, to see if new data has arrived. The reset function is used to resolve abnormal states, such as getting the connection back in synchronization after a break operation has occurred.

2.3.9 Net8 and the Transparent Network Substrate (TNS)

Net8 uses the Transparent Network Substrate and industry-standard networking protocols to accomplish its basic functionality. TNS is a foundation technology that is built into Net8 providing a single, common interface to all industry-standard protocols.

With TNS, peer-to-peer application connectivity is possible where no direct machine-level connectivity exists. In a peer-to-peer architecture, two or more computers can communicate with each other directly, without the need for any intermediary devices. In a peer-to-peer system, a node can be both a client and a server.

2.3.10 Net8 Architecture

Oracle networking environments are based on two concepts.

2.3.10.1 Distributed Processing

Oracle databases and client applications operate in what is known as a distributed processing environment. Distributed or cooperative processing involves interaction between two or more computers to complete a single data transaction. Applications such

as an Oracle tool act as clients requesting data to accomplish a specific operation. Database servers store and provide the data.

In a typical network configuration, clients and servers may exist as separate logical entities on separate physical machines. This configuration allows for a division of labor where resources are allocated efficiently between a client workstation and the server machine. Clients normally reside on desktop computers with just enough memory to execute user friendly applications, while a server has more memory, disk storage, and processing power to execute and administer the database.

This type of client-server architecture also enables you to distribute databases across a network. A distributed database is a network of databases stored on multiple computers that appears to the user as a single logical database. Distributed database servers are connected by a database link, or path from one database to another. One server uses a database link to query and modify information on a second server as needed, thereby acting as a client to the second server.

2.3.10.2 Stack Communications

The concept of distributed processing relies on the ability of computers separated by both design and physical location to communicate and interact with each other. This is accomplished through a process known as stack communications.

Stack communications can be explained by referencing the Open System Interconnection model. In the OSI model, communication between separate computers occurs in a stack-like fashion with information passing from one node to the other through several layers of code. Figure 2-5 depicts a typical OSI Protocol Communications Stack.

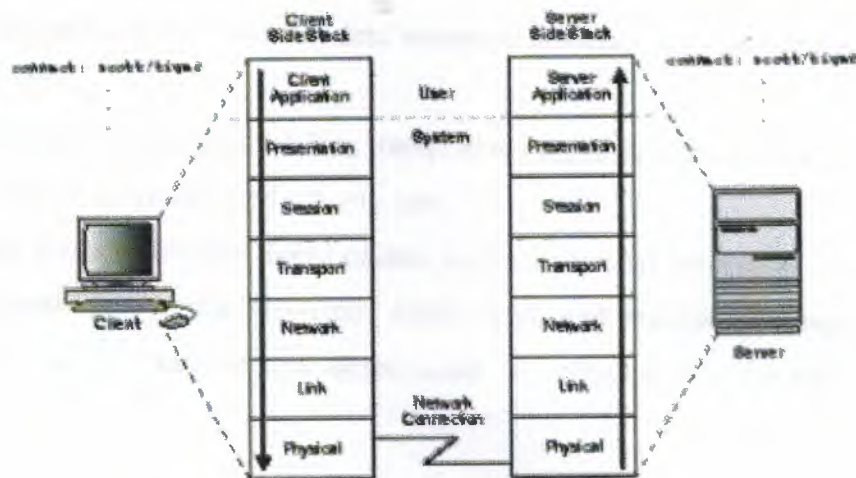


Figure 2-5 OSI Communications Stack

Information descends through layers on the client side where it is packaged for transport across a network medium in a manner that it can be translated and understood by corresponding layers on the server side. A typical OSI protocol communications stack will contain seven such layers.

1. **Application:** this is the OSI layer closest to the user, and as such is dependent on the functionality requested by the user. For example, in a database environment, a Forms application may attempt to initiate communication in order to access data from a server.
2. **Presentation:** ensures that information sent by the application layer of one system is readable by the application layer of another system. This includes keeping track of syntax and semantics of the data transferred between the client and server. If necessary, the presentation layer translates between multiple data representation formats by using a common data format.
3. **Session:** as its name suggests, establishes, manages, and terminates sessions between the client and server. This is a virtual pipe that carries data requests and responses. The session layer manages whether the data traffic can go in both directions at the same time referred to as asynchronous, or in only one direction at a time referred to as synchronous.

4. **Transport:** implements the data transport ensuring that the data is transported reliably.
5. **Network:** ensures that the data transport is routed through optimal paths through a series of interconnected subnetworks.
6. **Link:** provides reliable transit of data across a physical link.
7. **Physical:** defines the electrical, mechanical, and procedural specifications for activating, maintaining and deactivating the physical link between client and server.

2.3.10.3 Stack Communications in an Oracle networking environment

Stack communications allow Oracle clients and servers to share, modify, and manipulate data between themselves. The layers in a typical Oracle communications stack are similar to those of a standard OSI communications stack.

2.3.10.3.1 Client-Server Interaction

In an Oracle client-server transaction, information passes through the following layers

1. Client Application.
2. Oracle Call Interface.
3. Two Task Common.
4. Net8.
5. Oracle Protocol Adapters.
6. Network Specific Protocols.

Figure 2.6 depicts a typical communications stack in an Oracle networking environment.

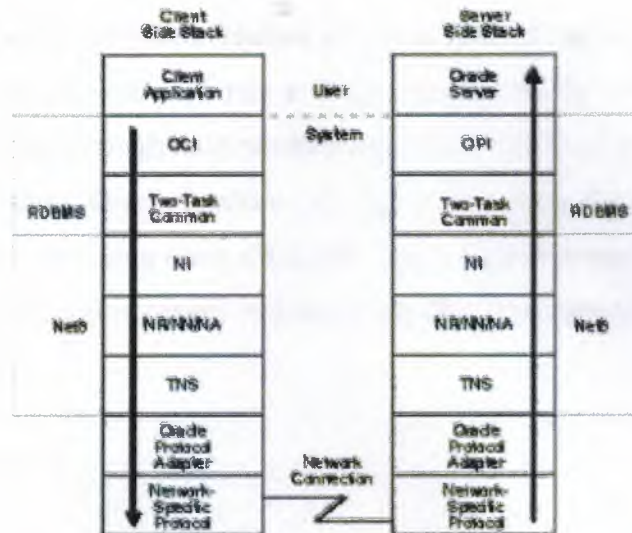


Figure2.6 Typical Communications Stack in an Oracle environment

1. Client Application

Oracle client applications provide all user-oriented activities, such as character or graphical user display, screen control, data presentation, application flow, and other application specifics. The application identifies database operations to send to the server and passes them through to the Oracle Call Interface.

2. Oracle Call Interface (OCI)

The OCI code contains all the information required to initiate a SQL dialogue between the client and the server. It defines calls to the server to:

1. Parse SQL statements for syntax validation.
2. Open a cursor for the SQL statement.
3. Bind client application variables into the server shared memory.
4. Describe the contents of the fields being returned based on the values in the server's data dictionary.
5. Execute SQL statements within the cursor memory space.
6. Fetch one or more rows of data into the client application.
7. Close the cursor.

The client application uses a combination of these calls to request activity within the server. OCI calls can be combined into a single message to the server, or they may be processed one at a time through multiple messages to the server, depending on the nature of the client application. Oracle products attempt to minimize the number of messages sent to the server by combining many OCI calls into a single message to the server. When a call is performed, control is passed to Net8 to establish the connection and transmit the request to the server.

3. Two-Task Common

Two-Task Common provides character set and data type conversion between different character sets or formats on the client and server. This layer is optimized to perform conversion only when required on a per connection basis.

At the time of initial connection, Two Task Common is responsible for evaluating differences in internal data and character set representations and determining whether conversions are required for the two computers to communicate.

4. Net8

Net8 provides all session layer functionality in an Oracle communications stack. It is responsible for establishing and maintaining the connection between a client application and server, as well as exchanging messages between them. Net8 itself has three component layers that facilitate session layer functionality.

1. **Network Interface:** This layer provides a generic interface for Oracle clients, servers, or external processes to access Net8 functions. The NI handles the break and reset requests for a connection.
2. **Network Routing/ Network Naming/ Network Authentication:** NR provides routing of the session to the destination. This may include any intermediary destinations or 'hops', on the route to the server destination. NN resolves aliases to a Net8 destination address. NA negotiates any authentication requirement with the destination.

3. **Transparent Network Substrate:** TNS is an underlying layer of Net8 providing a common interface to industry standard protocols. TNS receives requests from Net8, and settles all generic machine-level connectivity issues, such as the location of the server or destination, whether one or more protocols will be involved in the connection, and how to handle interrupts between client and server based on the capabilities of each. The generic set of TNS functions passes control to an Oracle Protocol Adapter to make a protocol-specific call. Additionally, TNS supports encryption and sequenced cryptographic message digests to protect data in transit.

5. Oracle Protocol Adapters

Oracle Protocol Adapters are responsible for mapping TNS functionality to industry-standard protocols used in the client-server connection. Each adapter is responsible for mapping the equivalent functions between TNS and a specific protocol.

6. Network-Specific Protocols

All Oracle software in the client-server connection process require an existing network protocol stack to make the machine-level connection between the two machines. The network protocol is responsible only for getting the data from the client machine to the server machine, at which point the data is passed to the server-side Oracle Protocol Adapter.

7. Server-Side Interaction

Information passed from a client application across a network protocol is received by a similar communications stack on the server side. The process stack on the server side is the reverse of what occurred on the client side with information ascending through communication layers. The one operation unique to the server side is the act of receiving the initial connection through the network listener.

The following components above the Net8 session layer are different from those on the client side.

1. Oracle Program Interface
2. Oracle Server

1. Oracle Program Interface

The OPI performs a complementary function to that of the OCI. It is responsible for responding to each of the possible messages sent by the OCI. For example, an OCI request to fetch 25 rows would have an OPI response to return the 25 rows once they have been fetched.

2. Oracle Server

The Oracle Server side of the connection is responsible for receiving dialog requests from the client OCI code and resolving SQL statements on behalf of the client application. Once received, a request is processed and the resulting data is passed to the OPI for responses to be formatted and returned to the client application.

2.3.10.4 Server-to-Server Interaction

When two servers communicate to complete a distributed transaction, the process, layers, and dialogues are the same as in the client-server scenario, except that there is no client application. The server has its own version of OCI, called the Network Program Interface (NPI). The NPI interface performs all of the functions that the OCI does for clients, allowing a coordinating server to construct SQL requests for additional servers.

2.4 DISTRIBUTED COMPUTING USING JAVA

Distributed systems require that computations running in different address spaces, potentially on different hosts, be able to communicate. For a basic communication mechanism, the JavaTM language supports sockets, which are flexible and sufficient for general communication. However, sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone.

An alternative to sockets is Remote Procedure Call, which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packaged up and shipped off to the remote target of the call. RPC systems encode arguments and return values using an external data representation.

RPC, however, does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require remote method invocation or RMI. In such systems, a local surrogate object manages the invocation on a remote object.

2.4.1 Distributed Object Applications

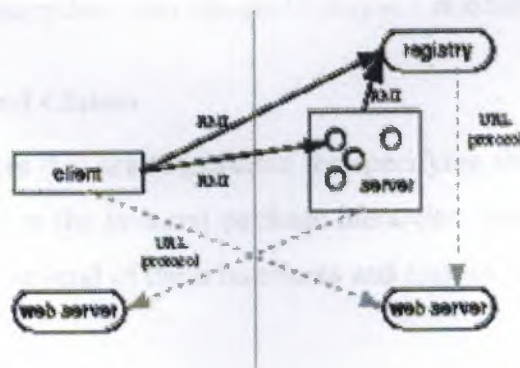
RMI applications are often comprised of two separate programs, a server and a client. A typical server application creates a number of remote objects, makes references to those remote objects accessible, and waits for clients to invoke methods on those remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

2.4.2 The Distributed and Nondistributed Models Contrasted

Distributed object applications need to:

1. Locate remote objects: Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility, the rmiregistry, or the application can pass and return remote object references as part of its normal operation.
2. Communicate with remote objects: Details of communication between remote objects are handled by RMI, to the programmer, remote communication looks like a standard Java method invocation.
3. Load class bytecodes for objects that are passed as parameters or return values: Because RMI allows a caller to pass pure Java objects to remote objects, RMI provides the necessary mechanisms for loading an object's code as well as transmitting its data.

The illustration below depicts an RMI distributed application that uses the registry to obtain references to a remote object. The server calls the registry to associate a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load Java class bytecodes, from server to client and from client to server, for objects when needed. RMI can load class bytecodes using any URL protocol that is supported by the Java system.



2.7 The Distributed and Nondistributed Models Contrasted

The Java distributed object model is similar to the Java object model in the following ways:

1. A reference to a remote object can be passed as an argument or returned as a result in any method invocation.
2. A remote object can be cast to any of the set of remote interfaces supported by the implementation using the built-in Java syntax for casting.
3. The built-in Java instanceof operator can be used to test the remote interfaces supported by a remote object.

The Java distributed object model differs from the Java object model in these ways:

1. Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces.
2. Non-remote arguments to, and results from, a remote method invocation are passed by copy rather than by reference. This is because references to objects are only useful within a single virtual machine.
3. A remote object is passed by reference, not by copying the actual remote implementation.
4. The semantics of some of the methods defined by class `java.lang.Object` are specialized for remote objects.
5. Since the failure modes of invoking remote objects are inherently more complicated than the failure modes of invoking local objects, clients must deal with additional exceptions that can occur during a remote method invocation.

2.4.3 RMI Interfaces and Classes

The interfaces and classes that are responsible for specifying the remote behavior of the RMI system are defined in the `java.rmi` package hierarchy. The following figure shows the relationship between several of these interfaces and classes.

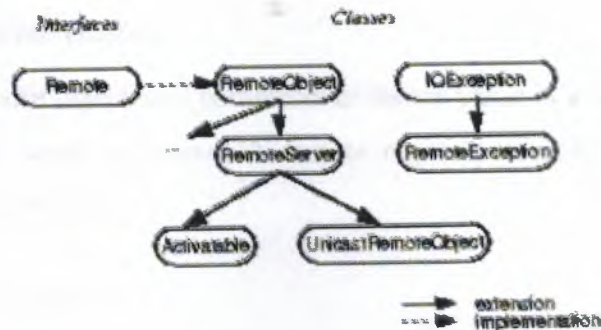


Figure 2.8 RMI Interfaces and Classes

2.4.3.1 The `java.rmi.Remote` Interface

In RMI, a *remote* interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine. In a remote method declaration, a remote object declared as a parameter or return value must be declared as the remote *interface*, not the implementation class of that interface. The interface `java.rmi.Remote` is a marker interface that defines no methods. A remote interface must *at least* extend the interface `java.rmi.Remote` or another remote interface that extends `java.rmi.Remote`.

2.4.4 Parameter Passing in Remote Method Invocation

An argument to, or a return value from, a remote object can be any Java object that is *serializable*. This includes Java primitive types, remote Java objects, and non-remote Java objects that implement the `java.io.Serializable` interface.

2.4.4.1 Passing Non-remote Objects

A non-remote object, that is passed as a parameter of a remote method invocation or returned as a result of a remote method invocation, is passed by *copy*; that is, the object is serialized using the Java Object Serialization mechanism.

So, when a non-remote object is passed as an argument or return value in a remote method invocation, the content of the non-remote object is copied before invoking the call on the remote object. When a non-remote object is returned from a remote method invocation, a new object is created in the calling virtual machine.

2.4.4.2 Passing Remote Objects

When passing a remote object as a parameter or return value in a remote method call, the stub for the remote object is passed. A remote object passed as a parameter can only implement remote interfaces.

2.4.4.3 Referential Integrity

If two references to an object are passed from one Virtual Machine to another Virtual Machine in parameters in a single remote method call and those references refer to the same object in the sending Virtual Machine, those references will refer to a single copy of the object in the receiving Virtual Machine. Within a single remote method call, the RMI system maintains referential integrity among the objects passed as parameters or as a return value in the call.


2.4.4.4 Class Annotation

When an object is sent from one Virtual Machine to another in a remote method call, the RMI system annotates the class descriptor in the call stream with the URL information of the class so that the class can be loaded at the receiver. It is a requirement that classes be downloaded on demand during remote method invocation.

2.4.4.5 Parameter Transmission

Parameters in an RMI call are written to a stream that is a subclass of the class `java.io.ObjectOutputStream` in order to serialize the parameters to the destination of the remote call. The `ObjectOutputStream` subclass overrides the `replaceObject` method to replace each remote object with its corresponding stub class. Parameters that are objects are written to the stream using the `ObjectOutputStream`'s `writeObject` method. The `ObjectOutputStream` calls the `replaceObject` method for each object written to the stream via the `writeObject` method. The `replaceObject` method of RMI's subclass of `ObjectOutputStream` returns the following:

1. If the object passed to `replaceObject` is an instance of `java.rmi.Remote`, then it returns the stub for the remote object. A stub for a remote object is obtained via a call to the method `java.rmi.server.RemoteObject.toStub`.

- 
2. If the object passed to `replaceObject` is not an instance of `java.rmi.Remote`, then the object is simply returned.

RMI's subclass of `ObjectOutputStream` also implements the `annotateClass` method that annotates the call stream with the location of the class so that it can be downloaded at the receiver.

Since parameters are written to a single `ObjectOutputStream`, references that refer to the same object at the caller will refer to the same copy of the object at the receiver. At the receiver, parameters are read by a single `ObjectInputStream`.

Any other default behavior of `ObjectOutputStream` for writing objects (and similarly `ObjectInputStream` for reading objects) is maintained in parameter passing. For example, the calling of `writeReplace` when writing objects and `readResolve` when reading objects is honored by RMI's parameter marshal and unmarshal streams.

In a similar manner to parameter passing in RMI as described above, a return value (or exception) is written to a subclass of `ObjectOutputStream` and has the same replacement behavior as parameter transmission.

2.4.5 Locating Remote Objects

A simple bootstrap name server is provided for storing named references to remote objects. A remote object reference can be stored using the URL-based methods of the class `java.rmi.Naming`.

For a client to invoke a method on a remote object, that client must first obtain a reference to the object. A reference to a remote object is usually obtained as a parameter or return value in a method call. The RMI system provides a simple bootstrap name server from which to obtain remote objects on given hosts. The `java.rmi.Naming` class provides Uniform Resource Locator (URL) based methods to look up, bind, rebind, unbind, and list the name-object pairings maintained on a particular host and port.

2.4.6 Stubs and Skeletons

RMI uses a standard mechanism for communicating with remote objects, stubs and skeletons. A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub which is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements.

When a stub's method is invoked, it does the following.

1. Initiates a connection with the remote VM containing the remote object.
2. Writes and transmits the parameters to the remote VM.
3. Waits for the result of the method invocation.
4. Reads the return value or exception returned.
5. Returns the value to the caller.

The stub hides the serialization of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote VM, each remote object may have a corresponding skeleton. The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following.

1. Reads the parameters for the remote method.
2. Invokes the method on the actual remote object implementation.
3. Writes and transmits the return value or exception to the caller.

2.4.7 Thread Usage in Remote Method Invocations

A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads. Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe.

2.4.8 Garbage Collection of Remote Objects

In a distributed system, just as in the local system, it is desirable to automatically delete those remote objects that are no longer referenced by any client. This frees the programmer from needing to keep track of the remote objects clients so that it can terminate appropriately. RMI uses a reference-counting garbage collection algorithm.

To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live references within each Java virtual machine. When a live reference enters a Java virtual machine, its reference count is incremented. The first reference to an object sends a referenced message to the server for the object. As live references are found to be unreferenced in the local virtual machine, the count is decremented. When the last reference has been discarded, an unreferenced message is sent to the server. Many subtleties exist in the protocol, most of these are related to maintaining the ordering of referenced and unreferenced messages in order to ensure that the object is not prematurely collected.

When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the Java virtual machine's garbage collector to discard the object if no other local references to the object exist. The distributed garbage collection algorithm interacts with the local Java virtual machine's garbage collector in the usual ways by holding normal or weak references to objects.

As long as a local reference to a remote object exists, it cannot be garbage-collected and it can be passed in remote calls or returned to clients. Passing a remote object adds the identifier for the virtual machine to which it was passed to the referenced set. A remote object needing unreferenced notification must implement the *java.rmi.server.Unreferenced* interface. When those references no longer exist, the unreferenced method will be invoked. unreferenced is called when the set of references is found to be empty so it might be called more than once. Remote objects are only collected when no more references, either local or remote, still exist.

Note that if a network partition exists between a client and a remote server object, it is possible that premature collection of the remote object will occur (since the transport might believe that the client crashed). Because of the possibility of premature collection, remote references cannot guarantee referential integrity; in other words, it is always possible that a remote reference may in fact not refer to an existing object. An attempt to use such a reference will generate a `RemoteException` which must be handled by the application.

2.4.9 Dynamic Class Loading

RMI allows parameters, return values and exceptions passed in RMI calls to be any object that is serializable. RMI uses the object serialization mechanism to transmit data from one virtual machine to another and also annotates the call stream with the appropriate location information so that the class definition files can be loaded at the receiver.

When parameters and return values for a remote method invocation are unmarshalled to become live objects in the receiving VM, class definitions are required for all of the types of objects in the stream. The unmarshalling process first attempts to resolve classes by name in its local class loading context. RMI also provides a facility for dynamically loading the class definitions for the actual types of objects passed as parameters and return values for remote method invocations from network locations specified by the transmitting endpoint. This includes the dynamic downloading of remote stub classes corresponding to particular remote object implementation classes as well as any other type that is passed by value in RMI calls, such as the subclass of a declared parameter type, that is not already available in the class loading context of the unmarshalling side.

To support dynamic class loading, the RMI runtime uses special subclasses of `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` for the marshal streams that it uses for marshalling and unmarshalling RMI parameters and return values. These subclasses override the `annotateClass` method of `ObjectOutputStream` and the `resolveClass` method of `ObjectInputStream` to communicate information about where to

locate class files containing the definitions for classes corresponding to the class descriptors in the stream.

For every class descriptor written to an RMI marshal stream, the `annotateClass` method adds to the stream the result of calling `java.rmi.server.RMIClassLoader.getClassAnnotation` for the class object, which may be null or may be a String object representing the codebase URL path from which the remote endpoint should download the class definition file for the given class.

For every class descriptor read from an RMI marshal stream, the `resolveClass` method reads a single object from the stream. If the object is a String, then `resolveClass` returns the result of calling `RMIClassLoader.loadClass` with the annotated String object as the first parameter and the name of the desired class in the class descriptor as the second parameter. Otherwise, `resolveClass` returns the result of calling `RMIClassLoader.loadClass` with the name of the desired class as the only parameter.

2.4.10 RMI Through Firewalls Via Proxies

The RMI transport layer normally attempts to open direct sockets to hosts on the Internet. Many intranets, however, have firewalls which do not allow this. The default RMI transport, therefore, provides two alternate HTTP-based mechanisms which enable a client behind a firewall to invoke a method on a remote object which resides outside the firewall.

2.4.10.1 How an RMI Call is Packaged within the HTTP Protocol

To get outside a firewall, the transport layer embeds an RMI call within the firewall-trusted HTTP protocol. The RMI call data is sent outside as the body of an HTTP POST request, and the return information is sent back in the body of the HTTP response. The transport layer will formulate the POST request in one of two ways.

1. If the firewall proxy will forward an HTTP request directed to an arbitrary port on the host machine, then it is forwarded directly to the port on which the RMI

server is listening. The default RMI transport layer on the target machine is listening with a server socket that is capable of understanding and decoding RMI calls inside POST requests.

2. If the firewall proxy will only forward HTTP requests directed to certain well-known HTTP ports, then the call will be forwarded to the HTTP server listening on port 80 of the host machine, and a CGI script will be executed to forward the call to the target RMI server port on the same machine.

2.4.10.2 The Default Socket Factory

The RMI transport extends the `java.rmi.server.RMISocketFactory` class to provide a default implementation of a socket factory which is the resource-provider for client and server sockets. This default socket factory creates sockets that transparently provide the firewall tunnelling mechanism as follows.

1. Client sockets automatically attempt HTTP connections to hosts that cannot be contacted with a direct socket.
2. Server sockets automatically detect if a newly-accepted connection is an HTTP POST request, and if so, return a socket that will expose only the body of the request to the transport and format its output as an HTTP response.

Client-side sockets, with this default behavior, are provided by the factory's `java.rmi.server.RMISocketFactory.createSocket` method. Server-side sockets with this default behavior are provided by the factory's `java.rmi.server.RMISocketFactory.createServerSocket` method.

2.4.10.3 Configuring the Client

There is no special configuration necessary to enable the client to send RMI calls through a firewall. The client can, however, disable the packaging of RMI calls as HTTP requests by setting the `java.rmi.server.disableHttp` property to equal the boolean value `true`.

2.4.10.4 Configuring the Server

In order for a client outside the server host's domain to be able to invoke methods on a server's remote objects, the client must be able to find the server. To do this, the remote references that the server exports must contain the fully-qualified name of the server host. Depending on the server's platform and network environment, this information may or may not be available to the Java virtual machine on which the server is running. If it is not available, the host's fully qualified name must be specified with the property `java.rmi.server.hostname` when starting the server.

2.4.10.5 Performance Issues and Limitations

Calls transmitted via HTTP requests are at least an order of magnitude slower than those sent through direct sockets, without taking proxy forwarding delays into consideration.

Because HTTP requests can only be initiated in one direction through a firewall, a client cannot export its own remote objects outside the firewall, because a host outside the firewall cannot initiate a method invocation back on the client.

CHAPTER 3

3.1 INTRODUCTION

The goal of this project is to make the process of student registration and maintenance in the university as easy as possible for the staff. Student registration is a very time consuming process, but by using a flexible software, we can reduce this headache. The life of the staff can be made easy by the following features:

- a) New student can be registered.
- b) Student details can be viewed.
- c) Student data can be deleted.
- d) Student data can be changed.
- e) The system is very easy to understand.
- f) The system is very secure.

3.2 PROGRAM IMPLEMENTATION

The program is divided into two parts: The database part and the application part. For the database, to meet the security and flexibility issues, I have chosen Oracle database which is one of the main topic of my overall project as well.

For the application part, java is my choice due to its object-oriented functionality and security features.

3.2.1 Database

To create the database, first I installed the Oracle 9i Server's Enterprise edition and configured it properly to function correctly. After basic installation, I created a database on Oracle Server. In the database, there are tables which hold the student record. Each table has primary and foreign key constraints and are related to each other using one-to-one and one-to-many relationships for the data integrity issues. To create the tables, there are three ways I have used time by time.

1. The easiest one is by using the table wizard in the Enterprise Manager Console. This wizard asks for the table fields and other important information and creates DDL by itself.
2. The second way is by using the SQL*Plus, here we need to write the SQL commands to create the table, specify the fields, specify if we want any constraints.
3. The third way I have used is by using a Java application. In the Java application, using appropriate code, connect to the database and then write the table definition. After compiling, run the program, the table will be created in the database.

Once the table has been created, its definition can be altered any time. To alter the structure of the table, we can implement any of the three ways discussed above.

3.2.2 Application

In the application, I have used the standard packages in Java to create user interface, connect to the database, event handling and other functions used in my application. The important packages used are:

1. java.awt.*
2. java.awt.event.*
3. javax.swing.*
4. java.sql.*
5. java.util.*
6. java.lang.*

Java's Abstract Windowing Toolkit and Swing packages are to create the user interface. These packages have all the components necessary to give the application a viewable look. These packages contain the Frame, TextField, TextArea, Button, Label, List box, Combo box, Check box, Option button, Horizontal and Vertical Scrollbar, Menu and Popup Menu components. The event handling mechanism in these packages is used to handle the events which occur during program execution and respond properly. All these

components and event handling mechanism have been used in the application to give the flexibility to the user to understand and use the application easily.

The java.sql package is used for database connectivity and manipulation functionality. To connect to the database and to do transactions we need to import this package.

This project is for student database where students data can be added, deleted, viewed and altered accurately with security. All the steps in the source code in Appendix A have been commented to understand the function of each step.

SUMMARY

The Oracle and Java combination in my application has resulted in a very useful and secure software. It's a complete Student Registration program. The user can register new, change, delete and view the student records. Although it is a complete program, but because of time limitations I could not expand it further. The program can be expanded to hold not only student basic record but as well as the course registration process, where the student courses could be managed in a suitable manner. The advisors will be able to see the courses for each student and register new courses with a few clicks of mouse. I hope in near future I will be able expand my application to fulfill all necessary requirements about student record in a university.

CONCLUSION

Oracle is a Relational Database Management Server designed to support the safe storage and manipulation of data. Oracle complements with a rich offering of well-integrated products like Java that is designed specifically for distributed processing and client/server applications. Oracle's database server has evolved to support Java for transaction processing and decision support to the extent that Oracle can provide a complete solution for client/server application development and deployment. We conclude that Oracle is a perfect platform for database design and processing with high level of security, but making some more developments in Java platform like increasing its execution speed can make it widely acceptable for enterprise applications.

APPENDIX A

A.1 Main.java

```
/*
 * The Main Menu Class, In this class we have four command Buttons for
 * Register, Change, Delete and View Student Record. We have a File
 * Menu with the same options.
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Main extends JFrame implements ActionListener{

    //Menu Declarations.
    private JMenuItem menuHelpAbout = new JMenuItem();
    private JMenu menuHelp = new JMenu();
    private JMenuItem menuNew = new JMenuItem();
    private JMenu menuFile = new JMenu();
    private JMenuBar menuBar = new JMenuBar();
    private JMenuItem menuChange = new JMenuItem();
    private JMenuItem menuDelete = new JMenuItem();
    private JMenuItem menuView = new JMenuItem();
    private JMenuItem menuExit = new JMenuItem();

    //Button Declarations.
    private JButton cmdNew = new JButton();
    private JButton cmdChange = new JButton();
    private JButton cmdDelete = new JButton();
    private JButton cmdView = new JButton();
    private JButton cmdExit = new JButton();

    //The constructor starts here.
    public Main(){
        try{
            jbInit();
            setSize(500,400);
            setTitle("Main Menu");
            show();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

```
private void jbInit() throws Exception{
```

```
    this.setJMenuBar(menuBar);
    this.getContentPane().setLayout(null);
    this.setSize(new Dimension(447, 363));
    menuFile.setText("File");
    menuChange.setText("Change");
    menuChange.setForeground(Color.blue);
    menuChange.addActionListener(this);
    menuDelete.setText("Delete");
    menuDelete.setForeground(Color.blue);
    menuView.setText("View All");
    menuView.setForeground(Color.blue);
    menuExit.setText("Exit");
    menuExit.setForeground(Color.red);
    cmdNew.setText("Register New Student");
    cmdNew.setBounds(new Rectangle(140, 35, 165, 35));
    cmdNew.setBackground(Color.lightGray);
    cmdNew.setBorder(BorderFactory.createLineBorder(Color.green, 1));
    cmdChange.setText("Change Student Details");
    cmdChange.setBounds(new Rectangle(140, 90, 165, 35));
    cmdChange.setBackground(Color.lightGray);
    cmdChange.setBorder(BorderFactory.createLineBorder(Color.green, 1));
    cmdDelete.setText("Delete Student Details");
    cmdDelete.setBounds(new Rectangle(140, 145, 165, 40));
    cmdDelete.setBackground(Color.lightGray);
    cmdDelete.setBorder(BorderFactory.createLineBorder(Color.green, 1));
    cmdView.setText("View All Student Record");
    cmdView.setBounds(new Rectangle(140, 205, 165, 40));
    cmdView.setBackground(Color.lightGray);
    cmdView.setBorder(BorderFactory.createLineBorder(Color.green, 1));
    cmdExit.setText("Exit");
    cmdExit.setBounds(new Rectangle(320, 265, 70, 30));
    cmdExit.setBackground(Color.lightGray);
    cmdExit.setBorder(BorderFactory.createLineBorder(Color.red, 1));

    cmdNew.addActionListener(this);
    cmdChange.addActionListener(this);
    cmdDelete.addActionListener(this);
    cmdView.addActionListener(this);
    cmdExit.addActionListener(this);
    menuDelete.addActionListener(this);
    menuView.addActionListener(this);
    menuExit.addActionListener(this);
```

```

menuNew.setText("New");
menuNew.setForeground(Color.blue);
menuNew.addActionListener(this);
menuHelp.setText("Help");
menuHelpAbout.setText("About");
menuHelpAbout.setForeground(Color.blue);
menuHelpAbout.addActionListener(this);
menuFile.add(menuNew);
menuFile.add(menuChange);
menuFile.add(menuDelete);
menuFile.add(menuView);
menuFile.add(menuExit);
menuBar.add(menuFile);
menuHelp.add(menuHelpAbout);
menuBar.add(menuHelp);
this.getContentPane().add(cmdExit, null);
this.getContentPane().add(cmdView, null);
this.getContentPane().add(cmdDelete, null);
this.getContentPane().add(cmdChange, null);
this.getContentPane().add(cmdNew, null);
}

```

//Event handling in actionPerformed method.

```

public void actionPerformed(ActionEvent ae){

```

```

    if(ae.getSource()==cmdNew)
        new StudentNew();

    if(ae.getSource()==menuNew)
        new StudentNew();

    if(ae.getSource()==cmdChange)
        new StudentChange();

    if(ae.getSource()==menuChange)
        new StudentChange();

    if(ae.getSource()==cmdDelete)
        new StudentDelete();

    if(ae.getSource()==menuDelete)
        new StudentDelete();

    if(ae.getSource()==menuView)
        new StudentView();

```



```

        if(ae.getSource()==cmdView)
            new StudentView();

        if(ae.getSource()==cmdExit)
            System.exit(0);

        if(ae.getSource()==menuExit)
            System.exit(0);
    }

    //main method.
    public static void main(String args[]){

        Main app=new Main();

        app.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }
}

```

A.2 StudentNew.java

```

/*****
 * StudentNew class, In this class there are three TextFields to store Student Number, *
 * FirstName, LastName fields and there are two Combo Boxes to store Gender and *
 * Department fields. There are five Labels and one command Button as well. *
 * In the constructor of the class I am connecting to the database and when the *
 * user will click OK button, the data will be added to database. *
 *****/

import java.awt.*;
import javax.swing.border.BevelBorder;
import java.awt.event.*;
import java.sql.*;
import javax.swing.*;

public class StudentNew extends JFrame implements ActionListener{

    private JButton cmdRegister = new JButton();
    private JTextField txtStNo = new JTextField();
    private JTextField txtFirstName = new JTextField();

```

```

private JTextField txtLastName = new JTextField();
private JTextField txtGender = new JTextField();
private JTextField txtDepartment = new JTextField();
private JLabel jLabel1 = new JLabel();
private JLabel jLabel2 = new JLabel();
private JLabel jLabel3 = new JLabel();
private JLabel jLabel4 = new JLabel();
private JLabel jLabel5 = new JLabel();
private Connection connect;
String url;
Choice cmbGender=new Choice();
Choice cmbDepartment=new Choice();

public StudentNew(){
    try{
        jbInit();
        setSize(500,400);
        setTitle("New Registration");
        show();

        //setup database connection.
        try{
            url="jdbc:oracle:thin:@mycomWin2000:1521:db1";
            String user="scott";
            String passw="tiger";
            Class.forName("oracle.jdbc.driver.OracleDriver");
            connect=DriverManager.getConnection(url,user,passw);
        }
        catch(ClassNotFoundException cnfex){
            cnfex.printStackTrace();
            JOptionPane.showMessageDialog(null,"Could not connect to the Database");
        }
        catch(SQLException sqllex){
            sqllex.printStackTrace();
            JOptionPane.showMessageDialog(null,"Could not connect to the Database");
        }
        catch(Exception ex){
            ex.printStackTrace();
            JOptionPane.showMessageDialog(null,"Could not connect to the Database");
        }
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

```

```
private void jbInit() throws Exception{
```

```
    this.getContentPane().setLayout(null);
    this.setSize(new Dimension(493, 370));
    cmdRegister.setText("Register");
    cmdRegister.addActionListener(this);
    cmdRegister.setBounds(new Rectangle(235, 270, 100, 35));
    cmdRegister.setToolTipText("Click Here to Resgister the Student.");
    txtStNo.setBounds(new Rectangle(195, 75, 115, 25));
    txtStNo.setBorder(BorderFactory.createLineBorder(Color.cyan, 1));
    txtStNo.setForeground(SystemColor.desktop);
    txtFirstName.setBounds(new Rectangle(195, 105, 115, 25));
    txtFirstName.setBorder(BorderFactory.createLineBorder(Color.cyan, 1));
    txtFirstName.setForeground(SystemColor.desktop);
    txtLastName.setBounds(new Rectangle(195, 135, 115, 25));
    txtLastName.setBorder(BorderFactory.createLineBorder(Color.cyan, 1));
    txtLastName.setForeground(SystemColor.desktop);
    cmbGender.add("Male");
    cmbGender.add("Female");
    cmbGender.setBounds(new Rectangle(195, 165, 115, 25));
    cmbGender.setForeground(SystemColor.desktop);
    cmbDepartment.add("Computer");
    cmbDepartment.add("Electrical");
    cmbDepartment.add("Mechanical");
    cmbDepartment.add("Civil");
    cmbDepartment.add("Business");
    cmbDepartment.setBounds(new Rectangle(195, 195, 115, 25));
    cmbDepartment.setForeground(SystemColor.desktop);
    jLabel1.setText("Student Number:");
    jLabel1.setBounds(new Rectangle(85, 75, 105, 25));
    jLabel1.setBackground(Color.black);
    jLabel1.setBorder(BorderFactory.createLineBorder(Color.green, 1));
    jLabel1.setForeground(SystemColor.desktop);
    jLabel2.setText("First Name:");
    jLabel2.setBounds(new Rectangle(85, 105, 105, 25));
    jLabel2.setBorder(BorderFactory.createLineBorder(Color.green, 1));
    jLabel2.setForeground(SystemColor.desktop);
    jLabel3.setText("Last Name:");
    jLabel3.setBounds(new Rectangle(85, 135, 105, 25));
    jLabel3.setBorder(BorderFactory.createLineBorder(Color.green, 1));
    jLabel3.setForeground(SystemColor.desktop);
    jLabel4.setText("Gender:");
    jLabel4.setBounds(new Rectangle(85, 165, 105, 25));
    jLabel4.setBorder(BorderFactory.createLineBorder(Color.green, 1));
    jLabel4.setForeground(SystemColor.desktop);
    jLabel5.setText("Department:");
```



```

jLabel5.setBounds(new Rectangle(85, 195, 105, 25));
jLabel5.setBorder(BorderFactory.createLineBorder(Color.green, 1));
jLabel5.setForeground(SystemColor.desktop);
this.getContentPane().add(jLabel5, null);
this.getContentPane().add(jLabel4, null);
this.getContentPane().add(jLabel3, null);
this.getContentPane().add(jLabel2, null);
this.getContentPane().add(jLabel1, null);
this.getContentPane().add(cmbDepartment, null);
this.getContentPane().add(cmbGender, null);
this.getContentPane().add(txtLastName, null);
this.getContentPane().add(txtFirstName, null);
this.getContentPane().add(txtStNo, null);
this.getContentPane().add(cmdRegister, null);
}

public void actionPerformed(ActionEvent ae){

    if(ae.getSource()==cmdRegister){

        boolean flag=true;
        if(txtStNo.getText().length()==0){
            flag=false;
            JOptionPane.showMessageDialog(null,"Student Number Required!");
        }
        if(flag){
            if(txtFirstName.getText().length()==0){
                flag=false;
                JOptionPane.showMessageDialog(null,"Enter First Name, then Press Register.");
            }
        }

        if(flag){
            if(txtLastName.getText().length()==0){
                flag=false;
                JOptionPane.showMessageDialog(null,"Enter Last Name, then Press Register.");
            }
        }

        if(flag){
            try{
                Statement statement=connect.createStatement();

                String query="INSERT INTO
Student"+"StNo,FirstName,LastName,Gender,Department"+" values

```

```
("+txtStNo.getText()+"','"+txtFirstName.getText()+"','"+txtLastName.getText()+"','"+
cmbGender.getSelectedItem()+"','"+cmbDepartment.getSelectedItem()+"");
```

```
int result=statement.executeUpdate(query);
```

```
if(result==1){
```

```
JOptionPane.showMessageDialog(null,"Student Registered.");
```

```
txtStNo.setText("");
```

```
txtFirstName.setText("");
```

```
txtLastName.setText("");
```

```
txtGender.setText("");
```

```
txtDepartment.setText("");
```

```
}
```

```
else
```

```
JOptionPane.showMessageDialog(null,"Insertion failed");
```

```
statement.close();
```

```
}
```

```
catch(SQLException sqlex){
```

```
sqlex.printStackTrace();
```

```
//output.append(sqlex.toString());
```

```
}
```

```
}
```

```
}
```

```
}
```

```
public static void main(String args[]){
```

```
StudentNew app=new StudentNew();
```

```
app.addWindowListener(new WindowAdapter(){
```

```
public void windowClosing(WindowEvent e){
```

```
System.exit(0);
```

```
}
```

```
}
```

```
);
```

```
}
```

```
}
```

A.3 StudentChange.java

```

/*****
*           StudentChange class, In this class we search by Student ID first      *
*           The Student record will be displayed in the fields and changes can      *
*           be made. By pressing OK, the changes will be committed to the database  *
*****/

```

```

import java.awt.event.*;
import java.sql.*;
import javax.swing.*;
import java.awt.*;

public class StudentChange extends JFrame implements ActionListener{

    private JTextField txtStNo = new JTextField();
    private JTextField txtFirstName = new JTextField();
    private JTextField txtLastName = new JTextField();
    private JTextField txtGender = new JTextField();
    private JTextField txtDepartment = new JTextField();
    private JTextField txtSearch = new JTextField();

    private JButton cmdSearch = new JButton();
    private JLabel jLabel1 = new JLabel();
    private JLabel jLabel2 = new JLabel();
    private JLabel jLabel3 = new JLabel();
    private JLabel jLabel4 = new JLabel();
    private JLabel jLabel5 = new JLabel();
    private JLabel jLabel6 = new JLabel();
    private JButton cmdOK = new JButton();

    Connection connect;
    String url;

    public StudentChange(){
        try{
            this.getContentPane().setLayout(null);
            this.setSize(new Dimension(430, 365));
            txtStNo.setBounds(new Rectangle(190, 90, 140, 25));
            txtFirstName.setBounds(new Rectangle(190, 125, 140, 25));
            txtLastName.setBounds(new Rectangle(190, 160, 140, 25));
            txtGender.setBounds(new Rectangle(190, 195, 140, 25));
            txtDepartment.setBounds(new Rectangle(190, 230, 140, 25));
            txtSearch.setBounds(new Rectangle(145, 20, 120, 25));
            cmdSearch.setText("Search");
            cmdSearch.setBounds(new Rectangle(275, 20, 90, 25));
            cmdSearch.addActionListener(this);
            jLabel1.setText("Student Number:");
            jLabel1.setBounds(new Rectangle(65, 20, 80, 25));
            jLabel2.setText("Student Number:");
            jLabel2.setBounds(new Rectangle(100, 90, 100, 25));
            jLabel3.setText("First Name:");

```



```

jLabel3.setBounds(new Rectangle(125, 125, 105, 20));
jLabel4.setText("Last Name:");
jLabel4.setBounds(new Rectangle(125, 155, 95, 25));
jLabel5.setText("Gender:");
jLabel5.setBounds(new Rectangle(140, 195, 75, 20));
jLabel6.setText("Department:");
jLabel6.setBounds(new Rectangle(120, 230, 85, 25));
cmdOK.setText("OK");
cmdOK.setBounds(new Rectangle(280, 285, 70, 25));
this.getContentPane().add(cmdOK, null);
this.getContentPane().add(jLabel6, null);
this.getContentPane().add(jLabel5, null);
this.getContentPane().add(jLabel4, null);
this.getContentPane().add(jLabel3, null);
this.getContentPane().add(jLabel2, null);
this.getContentPane().add(jLabel1, null);
this.getContentPane().add(cmdSearch, null);
this.getContentPane().add(txtSearch, null);
this.getContentPane().add(txtDepartment, null);
this.getContentPane().add(txtGender, null);
this.getContentPane().add(txtLastName, null);
this.getContentPane().add(txtFirstName, null);
this.getContentPane().add(txtStNo, null);

txtStNo.setEditable(false);
txtFirstName.setEditable(false);
txtLastName.setEditable(false);
txtGender.setEditable(false);
txtDepartment.setEditable(false);
}
catch(Exception e){
    e.printStackTrace();
}

setSize(500,400);
setTitle("Data Change");
show();

//setup database connection.
try{
    url="jdbc:oracle:thin:@mycomWin2000:1521:db1";
    String user="scott";
    String passw="tiger";
    Class.forName("oracle.jdbc.driver.OracleDriver");
    connect=DriverManager.getConnection(url,user,passw);
}

```

```

        catch(ClassNotFoundException cnfex){
            cnfex.printStackTrace();
            JOptionPane.showMessageDialog(null,"Could not connect to the
            Database");
        }
        catch(SQLException sqlex){
            sqlex.printStackTrace();
            JOptionPane.showMessageDialog(null,"Could not connect to the
            Database");
        }
        catch(Exception ex){
            ex.printStackTrace();
            JOptionPane.showMessageDialog(null,"Could not connect to the
            Database");
        }
    }
}

```

```

public void actionPerformed(ActionEvent ae){

    if(ae.getSource()==cmdSearch){

        if(txtSearch.getText().length()==0){
            JOptionPane.showMessageDialog(null,"Enter Student Number to
            Search.");
        }

        if(txtSearch.getText().length()!=0){
            try{
                Statement statement=connect.createStatement();

                String query="Select * from Student "+
                    "where StNo= '"+
                    txtSearch.getText()+"'";

                ResultSet rs=statement.executeQuery(query);
                //display rs
                try{
                    rs.next();
                    int recordNumber=rs.getInt(1);
                    if(recordNumber!=0){
                        txtFirstName.setEditable(true);
                        txtLastName.setEditable(true);
                        txtGender.setEditable(true);
                        txtDepartment.setEditable(true);
                    }
                }
            }
        }
    }
}

```

```

        txtStNo.setText(String.valueOf(recordNumber));
        txtFirstName.setText(rs.getString(2));
        txtLastName.setText(rs.getString(3));
        txtGender.setText(rs.getString(4));
        txtDepartment.setText(rs.getString(5));
    }
    else
        JOptionPane.showMessageDialog(null, "No Record Found");
} catch (SQLException sqllex) {
    sqllex.printStackTrace();
    JOptionPane.showMessageDialog(null, "No Such Record Found.");
}
statement.close();
}
catch (SQLException sqllex) {
    sqllex.printStackTrace();
    //output.append(sqllex.toString());
}
cmdOK.addActionListener(this);
}
}

if(ae.getSource()==cmdOK){
try{
Statement statement1=connect.createStatement();

String query1="Update Student set "+
    "FirstName='"+txtFirstName.getText()+
    "',LastName='"+txtLastName.getText()+
    "',Gender='"+txtGender.getText()+
    "',Department='"+txtDepartment.getText()+
    "'where StNo='"+txtStNo.getText();

int result=statement1.executeUpdate(query1);
if(result==1)
    JOptionPane.showMessageDialog(null, "Changes Committed.");
else
    JOptionPane.showMessageDialog(null, "Could Not Update, Please Retry
    Later!");
statement1.close();
}
catch (SQLException sqllex) {
    sqllex.printStackTrace();
}
}
}

```



```

public static void main(String args[]){

    StudentChange app=new StudentChange();

    app.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
}

```

A.4 StudentDelete.java

```

/*****
 *      Student Delete class, In this class we search student by Student      *
 *      Number, display uneditable record and by clicking Delete Button      *
 *      the record will be deleted from the database.                        *
 *****/

import java.awt.event.*;
import java.sql.*;
import javax.swing.*;
import java.awt.*;

public class StudentDelete extends JFrame implements ActionListener{

    private JTextField txtStNo = new JTextField();
    private JTextField txtFirstName = new JTextField();
    private JTextField txtLastName = new JTextField();
    private JTextField txtGender = new JTextField();
    private JTextField txtDepartment = new JTextField();
    private JTextField txtSearch = new JTextField();

    private JButton cmdSearch = new JButton();
    private JLabel jLabel1 = new JLabel();
    private JLabel jLabel2 = new JLabel();
    private JLabel jLabel3 = new JLabel();
    private JLabel jLabel4 = new JLabel();
    private JLabel jLabel5 = new JLabel();
    private JLabel jLabel6 = new JLabel();
    private JButton cmdOK = new JButton();

```

```
Connection connect;  
String url;
```

```
public StudentDelete(){  
    try{  
        this.getContentPane().setLayout(null);  
        this.setSize(new Dimension(430, 365));  
        txtStNo.setBounds(new Rectangle(190, 90, 140, 25));  
        txtFirstName.setBounds(new Rectangle(190, 125, 140, 25));  
        txtLastName.setBounds(new Rectangle(190, 160, 140, 25));  
        txtGender.setBounds(new Rectangle(190, 195, 140, 25));  
        txtDepartment.setBounds(new Rectangle(190, 230, 140, 25));  
        txtSearch.setBounds(new Rectangle(145, 20, 120, 25));  
        cmdSearch.setText("Search");  
        cmdSearch.setBounds(new Rectangle(275, 20, 90, 25));  
        cmdSearch.addActionListener(this);  
        jLabel1.setText("Student Number:");  
        jLabel1.setBounds(new Rectangle(65, 20, 80, 25));  
        jLabel2.setText("Student Number:");  
        jLabel2.setBounds(new Rectangle(100, 90, 100, 25));  
        jLabel3.setText("First Name:");  
        jLabel3.setBounds(new Rectangle(125, 125, 105, 20));  
        jLabel4.setText("Last Name:");  
        jLabel4.setBounds(new Rectangle(125, 155, 95, 25));  
        jLabel5.setText("Gender:");  
        jLabel5.setBounds(new Rectangle(140, 195, 75, 20));  
        jLabel6.setText("Department:");  
        jLabel6.setBounds(new Rectangle(120, 230, 85, 25));  
        cmdOK.setText("OK");  
        cmdOK.setBounds(new Rectangle(280, 285, 70, 25));  
        this.getContentPane().add(cmdOK, null);  
        this.getContentPane().add(jLabel6, null);  
        this.getContentPane().add(jLabel5, null);  
        this.getContentPane().add(jLabel4, null);  
        this.getContentPane().add(jLabel3, null);  
        this.getContentPane().add(jLabel2, null);  
        this.getContentPane().add(jLabel1, null);  
        this.getContentPane().add(cmdSearch, null);  
        this.getContentPane().add(txtSearch, null);  
        this.getContentPane().add(txtDepartment, null);  
        this.getContentPane().add(txtGender, null);  
        this.getContentPane().add(txtLastName, null);  
        this.getContentPane().add(txtFirstName, null);  
        this.getContentPane().add(txtStNo, null);  
  
        txtStNo.setEditable(false);  
    }  
}
```

```

        txtFirstName.setEditable(false);
        txtLastName.setEditable(false);
        txtGender.setEditable(false);
        txtDepartment.setEditable(false);
    }
    catch(Exception e){
        e.printStackTrace();
    }

    setSize(500,400);
    setTitle("Data Delete");
    show();

    //setup database connection.
    try{
        url="jdbc:oracle:thin:@mycomWin2000:1521:db1";
        String user="scott";
        String passw="tiger";
        Class.forName("oracle.jdbc.driver.OracleDriver");
        connect=DriverManager.getConnection(url,user,passw);
    }
    catch(ClassNotFoundException cnfex){
        cnfex.printStackTrace();
        JOptionPane.showMessageDialog(null,"Could not connect to the
        Database");
    }
    catch(SQLException sqllex){
        sqllex.printStackTrace();
        JOptionPane.showMessageDialog(null,"Could not connect to the
        Database");
    }
    catch(Exception ex){
        ex.printStackTrace();
        JOptionPane.showMessageDialog(null,"Could not connect to the
        Database");
    }
}

public void actionPerformed(ActionEvent ae){

    if(ae.getSource()==cmdSearch){

        if(txtSearch.getText().length()==0){
            JOptionPane.showMessageDialog(null,"Enter Student Number to
            Search.");
        }
    }
}

```



```

if(txtSearch.getText().length()!=0){
try{
Statement statement=connect.createStatement();

String query="Select * from Student "+
"where StNo=" "+
txtSearch.getText()+"";
ResultSet rs=statement.executeQuery(query);
//display rs
try{
rs.next();
int recordNumber=rs.getInt(1);
if(recordNumber!=0){

txtStNo.setText(String.valueOf(recordNumber));
txtFirstName.setText(rs.getString(2));
txtLastName.setText(rs.getString(3));
txtGender.setText(rs.getString(4));
txtDepartment.setText(rs.getString(5));
}
else
JOptionPane.showMessageDialog(null,"No Record Found");
}catch(SQLException sqlex){
sqlex.printStackTrace();
JOptionPane.showMessageDialog(null,"No Such Record Found.");
}
statement.close();
}
catch(SQLException sqlex){
sqlex.printStackTrace();
//output.append(sqlex.toString());
}
cmdOK.addActionListener(this);
}
}

if(ae.getSource()==cmdOK){
try{
Statement statement1=connect.createStatement();
String query1="Delete * from Student"+
"where StNo=" "+
txtSearch.getText()+"";

int result=statement1.executeUpdate(query1);
if(result==1)
JOptionPane.showMessageDialog(null,"Changes Committed.");
}

```

```

        else
            JOptionPane.showMessageDialog(null, "Could Not Update, Please Retry
            Later!");
        statement1.close();
    }
    catch(SQLException sqlex){
        sqlex.printStackTrace();
    }
}
}
public static void main(String args[]){

    StudentDelete app=new StudentDelete();

    app.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
}
}

```

A.5 StudentView.java

```

/*****
 *   StudentView class, In this class by using JTable component, all the record
 *   of the students will be displayed. One combo box contains the index fields
 *   By selecting the index, we can sort out the data.
 *****/

import java.sql.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class StudentView extends JFrame implements ItemListener{

    private Connection connection;
    private JTable table;
    private JPanel topPanel;
    JLabel label1, label2, fill1, fill2, fill3;
    Choice os;
    String query;

```

```

public StudentView(){

    topPanel=new JPanel();
    topPanel.setLayout(new GridLayout(1,3));
    label1=new JLabel("Select Index here >>");
    topPanel.add(label1);
    os=new Choice();
    os.add("StNo");
    os.add("FirstName");
    os.add("LastName");
    os.add("Gender");
    os.add("Department");
    os.addItemListener(this);
    topPanel.add(os);
    label2=new JLabel(" to Sort Out!");
    topPanel.add(label2);

    Container c=getContentPane();
    c.setLayout(new BorderLayout());
    c.add(topPanel, BorderLayout.NORTH);

    setSize(500,400);
    show();

    //setup database connection.
    try{
        String url="jdbc:oracle:thin:@mycomWin2000:1521:db1";
        String user="scott";
        String passw="tiger";
        Class.forName("oracle.jdbc.driver.OracleDriver");
        connection=DriverManager.getConnection(url,user,passw);
    }
    catch(ClassNotFoundException cnfex){
        cnfex.printStackTrace();
        JOptionPane.showMessageDialog(null,cnfex.toString());
    }
    catch(SQLException sqlex){
        sqlex.printStackTrace();
        JOptionPane.showMessageDialog(null,sqlex.toString());
    }
    catch(Exception ex){
        ex.printStackTrace();
        JOptionPane.showMessageDialog(null,ex.toString());
    }
}

```



```

    query="Select * from Student";

    getTable();
}

private void getTable(){

    Statement statement;
    ResultSet resultset;

    try{
        statement=connection.createStatement();
        resultset=statement.executeQuery(query);
        displayResultSet(resultset);
        resultset.close();
        statement.close();
    }
    catch(SQLException sqllex){
        sqllex.printStackTrace();
    }
}

private void displayResultSet(ResultSet rs) throws SQLException{

    //position to first record.
    boolean moreRecords=rs.next();

    //if no record, display message.
    if(!moreRecords){
        JOptionPane.showMessageDialog(this,"No records to display");
        setTitle("No records to display");
        return;
    }

    setTitle("Student Record");

    Vector columnHeads=new Vector();
    Vector rows=new Vector();

    try{
        //get column heads.
        ResultSetMetaData rsmd=rs.getMetaData();

        for(int i=1;i<=rsmd.getColumnCount();i++)
            columnHeads.addElement(rsmd.getColumnName(i));

        //get row data
        do{

```

```

        rows.addElement(getNextRow(rs,rsmd));
    }while (rs.next());

    //display table with ResultSet contents.
    table=new JTable(rows,columnHeads);
    JScrollPane scroller=new JScrollPane(table);
    getContentPane().add(scroller, BorderLayout.CENTER);
    validate();
    }
    catch(SQLException sqlex){
        sqlex.printStackTrace();
    }
    }

    private Vector getNextRow(ResultSet rs, ResultSetMetaData rsmd) throws
        SQLException{

        Vector currentRow=new Vector();

        for(int i=1;i<=rsmd.getColumnCount();i++)
            currentRow.addElement(rs.getString(i));
        return currentRow;
    }

    public void shutDown(){
        try{
            connection.close();
        }
        catch(SQLException sqlex){
            System.err.println("Unable to disconnect");
            sqlex.printStackTrace();
        }
    }

    public void itemStateChanged(ItemEvent ie){

        query="select * from Student order by "+os.getSelectedItem()+"";
        getTable();
    }

    public static void main(String args[]){

        final StudentView app=new StudentView();

        app.addWindowListener(
            new WindowAdapter(){

```

```

public void windowClosing(WindowEvent e){
    app.shutdown();
    System.exit(0);
}

```

The above code is a simple example of a window closing event handler. It calls the `shutdown()` method of the `app` object and then calls `System.exit(0)` to terminate the program.

The `shutdown()` method is a method of the `app` object. It is a method that is used to shut down the application. The `System.exit(0)` method is a static method of the `System` class. It is used to terminate the program.

The `app` object is an object of the `app` class. It is a class that is used to represent the application.

The `System` class is a class that is used to represent the system. It is a class that is used to represent the system.

The `app` class is a class that is used to represent the application. It is a class that is used to represent the application.

The `System` class is a class that is used to represent the system. It is a class that is used to represent the system.

The `app` class is a class that is used to represent the application. It is a class that is used to represent the application.

The `System` class is a class that is used to represent the system. It is a class that is used to represent the system.

The `app` class is a class that is used to represent the application. It is a class that is used to represent the application.

The `System` class is a class that is used to represent the system. It is a class that is used to represent the system.

APPENDIX B

GLOSSARY OF JAVA AND RELATED TERMS

Abstract Window Toolkit (AWT)

A collection of graphical user interface (GUI) components that were implemented using native-platform versions of the components. These components provide that subset of functionality which is common to all native platforms. Largely supplanted by the Project Swing component set.

abstract

A Java(TM) programming language keyword used in a class definition to specify that a class is not to be instantiated, but rather inherited by other classes. An abstract class can have abstract methods that are not implemented in the abstract class, but in subclasses.

alpha value

A value that indicates the opacity of a pixel.

API

Application Programming Interface. The specification of how a programmer writing an application accesses the behavior and state of classes and objects.

applet

A component that typically executes in a Web browser, but can execute in a variety of other applications or devices that support the applet programming model.

argument

A data item specified in a method call. An argument can be a literal value, a variable, or an expression.

Bean

A reusable software component. Beans can be combined to create an application.

bit

The smallest unit of information in a computer, with a value of either 0 or 1.

bitwise operator

An operator that manipulates two values comparing each bit of one value to the corresponding bit of the other value.

block

In the Java(TM) programming language, any code between matching braces. Example: {
x = 1; }.

business logic

The code that implements the functionality of an application. In the Enterprise JavaBeans model, this logic is implemented by the methods of an enterprise bean.

byte

A sequence of eight bits. The Java(TM) programming language provides a corresponding byte type.

bytecode

Machine-independent code generated by the Java(TM) compiler and executed by the Java interpreter.

catch

A Java(TM) programming language keyword used to declare a block of statements to be executed in the event that a Java exception, or run time error, occurs in a preceding "try" block.

class

In the Java(TM) programming language, a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass will implicitly be `Object`.

class method

A method that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class.

classpath

A classpath is an environmental variable which tells the Java(TM) virtual machine* and Java technology-based applications (for example, the tools located in the JDK(TM) 1.1.X\bin directory) where to find the class libraries, including user-defined class libraries.

class variable

A data item associated with a particular class as a whole--not with particular instances of the class. Class variables are defined in class definitions.

client

In the client/server model of communications, the client is a process that remotely accesses resources of a compute server, such as compute power and large memory capacity.

codebase

Works together with the `code` attribute in the `<APPLET>` tag to give a complete specification of where to find the main applet class file: `code` specifies the name of the file, and `codebase` specifies the URL of the directory containing the file.

commit

The point in a transaction when all updates to any resources involved in the transaction are made permanent.

compilation unit

The smallest unit of source code that can be compiled. In the current implementation of the Java(TM) platform, the compilation unit is a file.

compiler

A program to translate source code into code to be executed by a computer. The Java(TM) compiler translates source code written in the Java programming language into bytecode for the Java virtual machine.

component

An application-level software unit supported by a container. Components are configurable at deployment time. The J2EE platform defines four types of components: enterprise beans, Web components, applets, and application clients.

constructor

A pseudo-method that creates an object. In the Java(TM) programming language, constructors are instance methods with the same name as their class. Constructors are invoked using the `new` keyword.

container

An entity that provides life cycle management, security, deployment, and runtime services to components. Each type of container (EJB, Web, JSP, servlet, applet, and application client) also provides component-specific services.

CORBA

Common Object Request Broker Architecture. A language independent, distributed object model specified by the Object Management Group (OMG).

declaration

A statement that establishes an identifier and associates attributes with it, without necessarily reserving its storage (for data) or providing the implementation

encapsulation

The localization of knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but as long as

the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.

enterprise bean

A component that implements a business task or business entity; either an entity beans or a session bean.

Enterprise JavaBeans(TM) (EJB)

A component architecture for the development and deployment of object-oriented, distributed, enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user and secure.

exception

An event during program execution that prevents the program from continuing normally; generally, an error. The Java(TM) programming language supports exceptions with the try, catch, and throw keywords. See also exception handler.

exception handler

A block of code that reacts to a specific type of exception. If the exception is for an error that the program can recover from, the program can resume executing after the exception handler has executed.

executable content

An application that runs from within an HTML file.

extends

Class X extends class Y to add functionality, either by adding fields or methods to class Y, or by overriding methods of class Y. An interface extends another interface by adding methods. Class X is said to be a subclass of class Y.

garbage collection

The automatic detection and freeing of memory that is no longer in use. The Java(TM) runtime system performs garbage collection so that programmers never explicitly free objects.

GUI

Graphical User Interface. Refers to the techniques involved in using graphics, along with a keyboard and a mouse, to provide an easy-to-use interface to some program.

HTML

HyperText Markup Language. This is a file format, based on SGML, for hypertext documents on the Internet. It is very simple and allows for the embedding of images, sounds, video streams, form fields and simple text formatting. References to other objects are embedded using URLs.

HTTP

HyperText Transfer Protocol. The Internet protocol, based on TCP/IP, used to fetch hypertext objects from remote hosts.

HTTPS

HTTP layered over the SSL protocol.

IDL

Interface Definition Language. APIs written in the Java(TM) programming language that provide standards-based interoperability and connectivity with CORBA (Common Object Request Broker Architecture).

IOP

Internet Inter-ORB Protocol. A protocol used for communication between CORBA object request brokers.

implements

A Java(TM) programming language keyword optionally included in the class declaration to specify any interfaces that are implemented by the current class.

instance

An object of a particular class. In programs written in the Java(TM) programming language, an instance of a class is created using the `new` operator followed by the class name.

interpreter

A module that alternately decodes and executes every statement in some body of code. The Java(TM) interpreter decodes and executes bytecode for the Java virtual machine

JAR Files (.jar)

Java ARchive. A file format used for aggregating many files into one.

Java(TM)

is Sun's trademark for a set of technologies for creating and safely running software programs in both stand-alone and networked environments.

Java Application Environment (JAE)

The source code release of the Java Development Kit (JDK(TM)) software.

Java Development Kit (JDK(TM))

A software development environment for writing applets and applications in the Java programming language.

Java(TM) Platform

Consists of the Java language for writing programs; a set of APIs, class libraries, and other programs used in developing, compiling, and error-checking programs; and a Java virtual machine which loads and executes the class files.

JavaScript(TM)

A Web scripting language that is used in both browsers and Web servers. Like all scripting languages, it is used primarily to tie other components together or to accept user input.

JavaServer Pages(TM) (JSP)

An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a Web browser.

Java(TM) virtual machine (JVM)

A software "execution engine" that safely and compatibly executes the byte codes in Java class files on a microprocessor (whether in a computer or in another electronic device).

Jini(TM) Technology

a set of Java APIs that may be incorporated an optional package for any Java 2 Platform Edition. The Jini APIs enable transparent networking of devices and services and eliminates the need for system or network administration intervention by a user.

The Jini technology is currently an optional package available on all Java platform editions.

JMAPI

Java(TM) Management API. A collection of Java programming language classes and interfaces that allow developers to build system, network, and service management applications.

JNDI

Java Naming and Directory Interface(TM). A set of APIs that assist with the interfacing to multiple naming and directory services.

JPEG

Joint Photographic Experts Group. An image file compression standard established by this group. It achieves tremendous compression at the cost of introducing distortions into the image which are almost always imperceptible.

JRE

Java(TM) runtime environment. A subset of the Java Developer Kit for end-users and developers who want to redistribute the runtime environment. The Java runtime

environment consists of the Java virtual machine*, the Java core classes, and supporting files.

Just-in-time (JIT) Compiler

A compiler that converts all of the bytecode into native machine code just as a Java(TM) program is run. This results in run-time speed improvements over code that is interpreted by a Java virtual machine*.

JVM

Java(TM) Virtual Machine*. The part of the Java Runtime Environment responsible for interpreting bytecodes.

multithreaded

Describes a program that is designed to have parts of its code execute concurrently.

SAX

Simple API for XML. An event-driven, serial-access mechanism for accessing XML documents.

Secure Socket Layer (SSL)

A protocol that allows communication between a Web browser and a server to be encrypted for privacy.

servlet

A Java program that extends the functionality of a Web server, generating dynamic content and interacting with Web clients using a request-response paradigm.

SQL

Structured Query Language. The standardized relational database language for defining database objects and manipulating data.

TCP/IP

Transmission Control Protocol based on IP. This is an Internet protocol that provides for the reliable delivery of streams of data from one host to another.

thread

The basic unit of program execution. A process can have several threads running concurrently, each performing a different job, such as waiting for events or performing a time-consuming job that the program doesn't need to complete before going on. When a thread has finished its job, the thread is suspended or destroyed.

throw

A Java(TM) programming language keyword that allows the user to throw an exception or any class that implements the "throwable" interface.

throws

A Java(TM) programming language keyword used in method declarations that specify which exceptions are not handled within the method but rather passed to the next higher level of the program.

try

A Java(TM) programming language keyword that defines a block of statements that may throw a Java language exception. If an exception is thrown, an optional "catch" block can handle specific exceptions thrown within the "try" block. Also, an optional "finally" block will be executed regardless of whether an exception is thrown or not.

URL

Uniform Resource Locator. A standard for writing a text reference to an arbitrary piece of data in the WWW. A URL looks like "protocol://host/localinfo" where protocol specifies a protocol to use to fetch the object (like HTTP or FTP), host specifies the Internet name of the host on which to find it, and localinfo is a string (often a file name) passed to the protocol handler on the remote host.

REFERENCES

Java 2, The Complete Reference

Fourth Edition

Herbert Schildt

Mc Graw Hill

Java Server Programming

J2EE Edition

Worx Press

Oracle 8i Application Programming with Java

Professional

Wrox Press

Java 2 Applets

Steven Holzner

bpb Press

Oracle JDeveloper

Cary Jenson

Mc Graw Hill

Java Programming

H.M. Deitel

Deitel Press