NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

Design and Programming in Java to Develop Security Measures on DBMS Integrated to Internet Platform

Graduation Project
COM 400

Students: Cem Uludağ (980482)

Supervisor: Halil Adahan

Lefkoşa - 2004

Acknowledgements

First of all, I would like to thank my supervisor Mr. Halil Adahan. Under his guidance, I successfully overcome many difficulties and learn a lot about Java Programming and Oracle Database and Graduation Project. When I faced problems in these fields, he always helped me.
I wish to thank my parents because of their endless support and encouragement and also my friends Hakan and Turgut Tuna brothers and my house friend Tunç Samurkaş for their help to develop myself by discussing technological subjects and computer science and also for their friendships.

It is my pleasure to take this opportunity to express my greatest gratitude to many individuals who have given me a lot of supports during my four-year Undergraduate program in the Near East University. Without them, my Graduation Project would not have been successfully completed on time may be never.

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. The basic components of database management systems are; collecting data and information, managing data storage, data retrieval and data update including insertion, modification and deletion. Database management systems require some automated tools for design, query and application development. Database applications and database users also have to be considered. Database users are seperated into four basic branches; database administrators, database designers, end users and application programmers. We can consider the 50's as the starting time of database management systems with the file systems. Then the evolution of DBMS is continued by hierarchical & network systems at 60's and 70's followed by relational systems at 80's and finally the object orinted systems are introduced at late 80's. With all of the steps in the evolution we have more advantages. The most well knowns are reduced data redundancy, data integrity, data independence, data sharing and data security.

The introduction of Java applets has taken theWorldWide Web by storm. Information servers can customize the presentation of their content with server-supplied code which executes inside the Web browser. We examine the Java language and both the HotJava and Netscape browsers which support it, and find a significant number of flaws which compromise their security. These flaws arise for several reasons, including implementation errors, unintended interactions between browser features, differences between the Java language and bytecode semantics, and weaknesses in the design of the language and the bytecode format. On a deeper level, these flaws arise because of weaknesses in the design methodology used in creating Java and the browsers. In addition to the flaws, we discuss the underlying tension between the openness desired by Web application writers and the security needs of their users,
and it is suggested how both might be accommodated.

Introduction

A database management system (DBMS) consists of a collection of interrelated data and a set of programs to access that data. The collection of data, usually refered to as the database, contains information about one particular enterprise. The primary goal of a DBMS is to provide an environment that is both convenient and efficient. To use in retrieving and storing database information.

Database systems are designed to manage large bodies of information. The management of data involves both the definition of structures for the storage of information and the provision of mechanisms for the manipulation of information. In addition, the database system must provide for the safety of the information stored, despite system crashes or attempts at unauthorized access. If the data is to be shared among several users, the system must avoid possible anomalous results.

The importance of information in most organizations, and hence the value of the database, has led to the development of a large body of concepts and techniques for the efficient management of data.

Table of Contents

List of Figures

Chapter I: Introduction

1.1 Java Platform

The Java platform consists of the Java application programming interfaces (APIs) and the Java virtual machine (JVM).

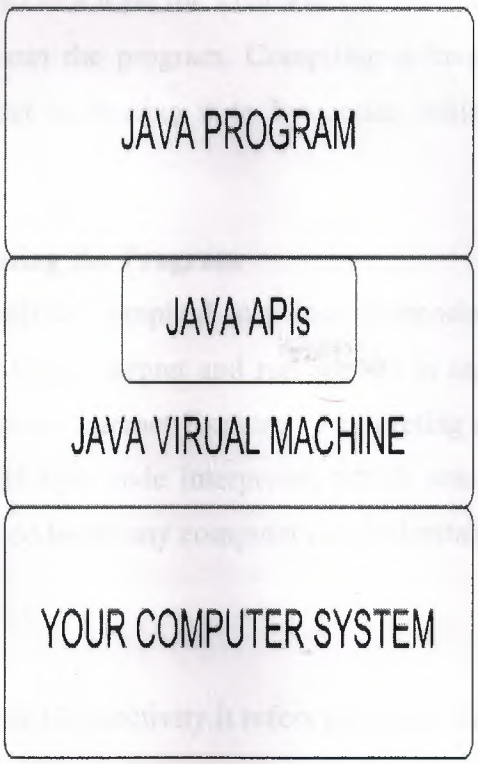

Figure 1.1.1 : The Java Platform

Java APIs are libraries of compiled code that we can use in your programs. They let you add ready-made and customizable functionality to save you programming time.

Java programs are run (or interpreted) by another program called the Java VM. Rather than running directly on the native operating system, the program is interpreted by the Java VM for the native operating system. This means that any computer system with the Java VM installed can run Java programs regardless of the computer system on which the applications were originally developed.

1.1.1 Writing a Program

The easiest way to write a simple program is with a text editor. So, using any text editor we can create a text file, and make the extension of the text file to .java. Java programs are case sensitive.

1.1.2 Compiling the Program

A program has to be converted to a form the Java VM can understand so any computer with a Java VM can interpret and run the program. Compiling a Java program means taking the programmer-readable text and converting it to bytecodes, which are platform-independent instructions for the Java VM.

1.1.3 Interpreting and Running the Program

After the source code succesfully compiled into Java bytecodes, we can interpret and run applications on any Java VM, or interpret and run applets in any Web browser with a Java VM built in such as Netscape or Internet Explorer. Interpreting and running a Java program means invoking the Java VM byte code interpreter, which converts the Java byte codes to platform-dependent machine codes so any computer can understand and run the program.

1.2 Database Access

1.2.1 The JDBC

JDBC stands for Java Database Connectivity.It refers to several things, depending on context:

1. It's a specification for using data sources in Java applets and applications.
2. It's an API for using low-level JDBC drivers.
3 It's an API for creating the low-level JDBC drivers, which do the actual connecting/transacting with data sources.
4. It's based on the X/Open SQL Call Level Interface (CLI) that defines how client/server interactions are implemented for database systems.

The JDBC defines every aspect of making data-aware Java applications and applets. The low-level JDBC drivers perform the database-specific translation to the high-level JDBC interface. This interface is used by the developer so we don't need to worry about the database-specific syntax when connecting to and querying different databases. The JDBC is a package, much like other Java packages such as java.awt. It's not currently a part of the standard Java Developer's Kit (JDK) distribution, but it is slated to be included as a standard part of the

general Java API as the java.sql package. The drivers necessary for connection to their respective databases do not require any pre-installation on the clients: A JDBC driver can be downloaded along with an applet.

1.2.2 The JDBC Structure

The JDBC is two-dimensional. The reasoning for the split is to separate the low-level programming from the high-level application interface. The low-level programming is the JDBC driver. The idea is that database vendors and third-party software vendors will supply pre-built drivers for connecting to different databases. JDBC drivers are quite flexible: They can be local data sources or remote database servers. The implementation of the actual connection to the data source/database is left entirely to the JDBC driver.

The structure of the JDBC includes these key concepts:

1. The goal of the JDBC is a DBMS independent interface, a "generic SQL database access framework," and a uniform interface to different data sources.
2. The programmer writes only *one* database interface; using JDBC, the program can access any data source without recoding.

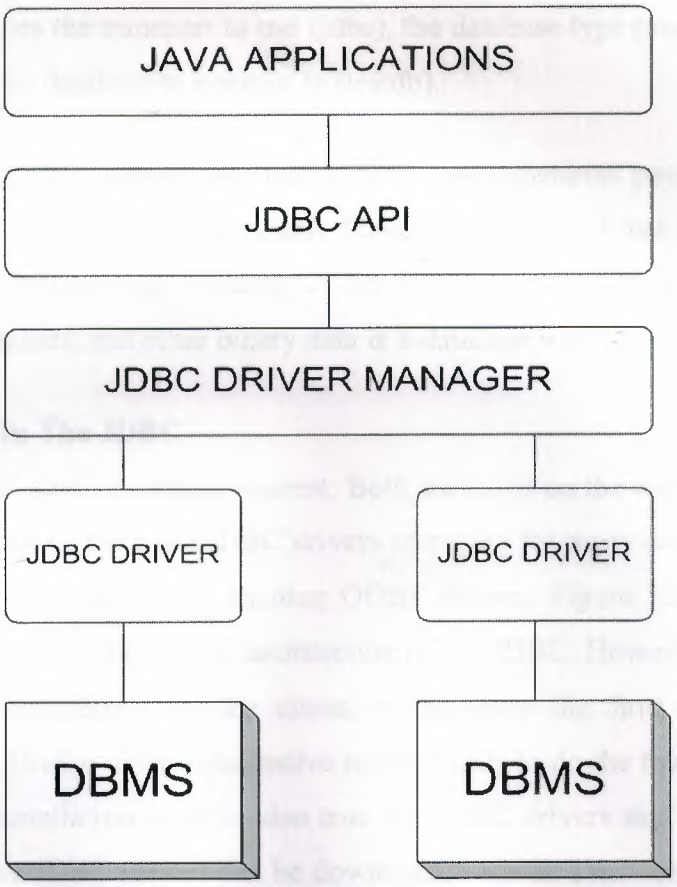

Figure 1.2.2.1 : The architecture of the JDBC

Figure 1.2.2 shows the architecture of the JDBC. The **DriverManager** class is used to open a connection to a database via a JDBC driver, which must register with the **DriverManager** before the connection can be formed. When a connection is attempted, the **DriverManager** chooses from a given list of available drivers to suit the explict type of database connection. After a connection is formed, the calls to query and fetch results are made directly with the JDBC driver. The JDBC driver must implement the classes to process these functions for the specific database, but the rigid specification of the JDBC ensures that the drivers will perform as expected. Essentially, the developer who has JDBC drivers for a certain database does not need to worry about changing the code for the Java program if a different type of database is used (assuming that the JDBC driver for the other database is available). This is especially useful in the scenario of distributed databases.

The JDBC uses a URL syntax for specifying a database. For example, a connection to a mSQL database:

jdbc:msql://mydatabase.server.com:1112/testdb

This statement specifies the transport to use (jdbc), the database type (msql), the server name, the port (1112), and the database to connect to (testdb).

The data types in SQL are mapped into native Java types whenever possible. When a native type is not present in Java, a class is available for retrieving data of that type. The JDBC also includes support for binary large objects, or BLOB data types; we can retreive and store images, sound, documents, and other binary data in a database with the JDBC.

1.2.3 ODBC's Part In The JDBC

The JDBC and ODBC share a common parent: Both are based on the same X/OPEN call level interface for SQL. Though there are JDBC drivers emerging for many databases, we can write database-aware Java programs using existing ODBC drivers. Figure 1.2 shows the place of the JDBC-ODBC Bridge in the overall architecture of the JDBC. However, the JDBC-ODBC Bridge requires pre-installation on the client, or wherever the Java program is actually running, because the Bridge must make native method calls to do the translation from ODBC to JDBC. This pre-installation issue is also true for JDBC drivers that use native methods. Only 100 percent Java JDBC drivers can be downloaded across a network with a Java applet, thus requiring no pre-installation of the driver.

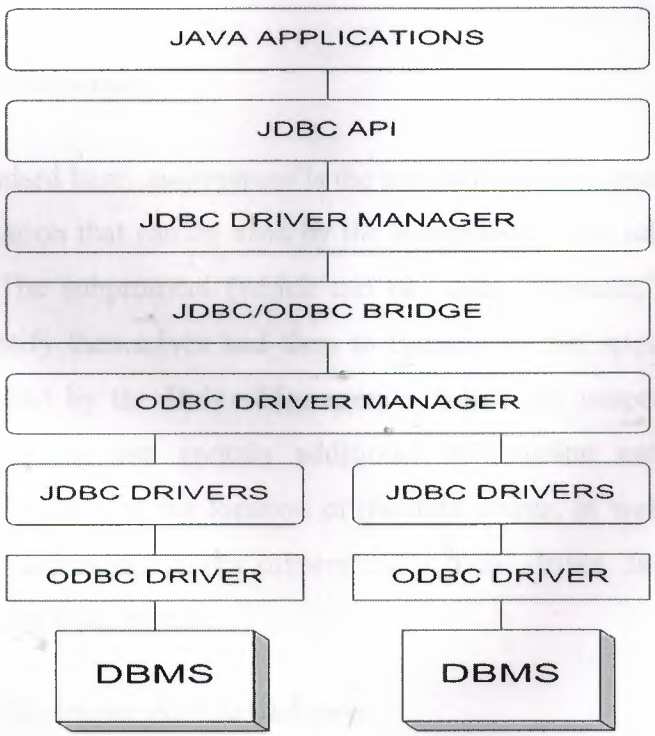

Figure 1.2.3.1 : ODBC in the JDBC model.

ODBC drivers function in the same manner as "true" JDBC drivers; in fact, the JDBC-ODBC bridge is actually a sophisticated JDBC driver that does low-level translation to and from ODBC. When the JDBC driver for a certain database becomes available, you can easily switch from the ODBC driver to the new JDBC driver with few, if any, changes to the code of the Java program.

1.2.4 Using JDBC Drivers

The first things we need to understand is how to use JDBC drivers and the JDBC API to connect to a data source.

There are no drivers packaged with the JDBC API so we must get them ourself from software vendors. If we want to use ODBC, we'll need ODBC drivers, as well. If we don't have a database server, but we want to use JDBC,we can use the ODBC drivers packaged with Microsoft Access. Using the JDBC-ODBC Bridge, we can write Java applications that can interact with an Access database.

1.2.5 JDBC URL And The Connection

The format for specifying a data source is an extended Universal Resource Locator (URL). The JDBC URL structure is broadly defined as follows;

jdbc:<subprotocol>:<subname>

where *jdbc* is the standard base, *subprotocol* is the particular data source type, and *subname* is an additional specification that can be used by the subprotocol. The subname is based solely on the subprotocol. The subprotocol (which can be "odbc," "oracle," etc.) is used by the JDBC drivers to identify themselves and then to connect to that specific subprotocol. The subprotocol is also used by the **DriverManager** to match the proper driver to a specific subprotocol. The subname can contain additional information used by the satisfying subprotocol (i.e. driver), such as the location of the data source, as well as a port number or catalog. Again, this is dependent on the subprotocol's JDBC driver. JavaSoft suggests that a network name follow the URL syntax:

jdbc:<subprotocol>://hostname:port/subsubname

The mSQL JDBC driver follows this syntax :

jdbc:msql://mycomputer.com:1112/databasename

The **DriverManager.getConnection** method in the JDBC API uses this URL when attempting to start a connection. A valid driver must be registered with the JDBC **DriverManager** before attempting to create this connection The **DriverManager.getConnection** method can be passed in a **Property** object where the keys "user," "password," and even "server" are set accordingly. The direct way of using the **getConnection** method involves passing these attributes in the constructor. The following is an example of how to create a **Connection** object from the **DriverManager.getConnection** method. This method returns a **Connection** object which is to be assigned to an instantiated **Connection** class:

```
String url="jdbc:msql://mydatabaseserver.com:1112/databasename";
Name = "pratik";
password = "";
Connection con;
con = DriverManager.getConnection(url, Name, password);
// remember to register the driver before doing this!
```

1.2.6 Using ODBC Drivers

In an effort to close the gap between existing ODBC drivers for data sources and the emerging pure Java JDBC drivers, JavaSoft and Intersolv released the JDBC-ODBC Bridge. Note that there is a Java interface (hidden as a JDBC driver called JdbcOdbcDriver and found in the jdbc/odbc/ directory) that does the necessary JDBC to ODBC translation with the native method library that is part of the JDBC-ODBC bridge package. Once the Bridge is set up, the JDBC handles access to the ODBC data sources just like access to normal JDBC drivers; in essence, we can use the same Java code with either JDBC drivers or ODBC drivers that use the Bridge—all we have to do is change the JDBC URL to reflect a different driver.

1.2.7 Bridge Requirements

JDBC-ODBC Bridge contains a very thin layer of native code. This library's sole purpose is to accept an ODBC call from Java, execute that call, and return any results back to the driver. All processing, including memory management, is contained within the Java side of the Bridge. Instead of being able to download Java class files and execute we must first install and configure additional software in order to use the Bridge. Required components are:

1. The Java Developer's Kit
2. The JDBC Interface classes (java.sql.*)
3. The JDBC-ODBC Bridge classes (jdbc.odbc.* or sun.jdbc.odbc.* for JDBC version 1.1 and higher)
4. An ODBC Driver Manager (such as the one provided by Microsoft for Win95/NT); do not confuse this with the JDBC DriverManager class
5. Any ODBC drivers to be used from the Bridge (from vendors such as Intersolv, Microsoft, and Visigenic)

1.2.8 The ODBC URL

To make a connection to a JDBC driver, we must supply a URL. The general structure of the JDBC URL is

jdbc:<subprotocol>:<subname>

where *subprotocol* is the kind of database connectivity being requested, and *subname* provides additional information for the subprotocol. For the Bridge, the specific URL structure is:

jdbc:odbc:<ODBC datasource name>[;attribute-name=attribute-value]...

The Bridge can only provide services for URLs that have a subprotocol of **odbc**. If a different subprotocol is given, the Bridge will simply tell the JDBC **DriverManager** that it has no idea what the URL means, and that it can't support it. The subname specifies the ODBC data source name to use, followed by any additional connection string attributes.

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.

1.3.1 Oracle Files

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

1.3.1.1 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.

1.3.1.2 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.

1.3.1.3 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.

1.3.1.3.1 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.

1.3.1.3.2 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.

1.3.1.4 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.

1.3.2 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.

1.3.2.1 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.

1.3.2.1.1 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.

1.3.2.1.2 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.

1.3.2.1.3 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.

1.3.2.1.4 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.

1.3.2.1.5 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.

1.3.2.2 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.

1.3.3 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.

1.3.3.1.1 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.

1.3.3.1.2 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.

1.3.3.1.3 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.

1.3.3.1.4 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.

1.3.3.1.5 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.

1.3.3.1.6 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.

1.3.3.2 Basic Translation Steps and Runtime Processing

1.3.3.2.1 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.
4. 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.
5. 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.
6. The JVM invokes the Java compiler, which is usually, but not necessarily, the standard javac provided with the Sun Microsystems JDK.
7. 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.
8. The JVM invokes the Oracle SQLJ customizer or other specified customizer.
9. The customizer customizes the profiles generated in step 4.

1.3.3.2.2 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.

1.4 Database

1.4.1 Introduction

A database management system (DBMS) consists of a collection of interrelated data and a set of programs to access that data. The collection of data , usually refered to as the database , contains information about one particular enterprise. The primary goal of a DBMS is to provide an environment that is both convenient and efficient. To use in retrieving and storing database information.

Database systems are designed to manage large bodies of information. The management of data involves both the definition of structures for the storage of information and the provision of mechanisms for the manipulation of information. In addition , the database system must provide for the safety of the information stored , despite system crashes or attempts at unauthorized access. If the data is to be shared among several users , the system must avoid possible unexpected results.

The importance of information in most organizations , and hence the value of the database , has led to the development of a large body of concepts and techniques for the efficient management of data.

In atypical file processing system , permanent records are stored in various files , and a number of different application programsa re written to extract records from and add records to the appropriate file. This scheme has a number of major disadvantages.

1. Data Redundancy and Inconsistency : Since the files and application programs are created by different programmers over a long period of time , the files are likely to have different formats and the programs may be written in several programming languages. Moerover , the same piece of information , may be duplicated in several files.

2. Difficulty in Accessing Data : Conventional file-processing environments do not allow data to be retrieved in a convenient and effiicient manner. Beter data retrieval systems must be developed for general use and applications.
3. Data Isolation : Since data is scattered in various files ,and files may be in different formats , it is difficult to write new application programs to retrieve the appropriate data.
4. Concurrent Access Anomalies : In order to improve the overall performance of the system and obtain a faster redponse time , many systems allow multiple users to update the data simultaneously. In such an environment , interaction of concurrent updates may result in consistent data. So a supervision must be maintained in the system. Since the data may be accessed by many different application programs which have noot been previously coordinated , supervision is very difficult to provide.
5. Security Problems : Not every user of the database system should be able to access all the data. Since application programs are added to the system in an ad hoc manner , it is difficult to enforce such security constraints.
6. Integrity problems : The data values stored in the database must satisfy certain types of consistency constraints. These constraints are enforced in the system by adding appropriate code in the various application programs. However , when new constraints are added , it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

1.4.2 Data Abstraction Mechanism

A DBMS is a collection of interrelated files and a set of programs that allows users to access and modify these files. A major purpose of database system is to provide usres with an abstract view of the data. That is, the system hides certain details of how the data is stored and maintained. However , in order for the system to be usable , data must be retrieved efficiently. This concern has lead to the design of complex data structures for the represantation of data in the database.

1. Physical Level: The lowest level of abstraction describe show the data are actually tored. (More information)
2. Conceptual Level: The next higher-level of abstraction describes what data are actually stored in database , and the relationships that exists among data. This level is

used by database administrators who must decide what information to be kept in the database.

3. View Level: The highest level of abstraction describes only the part of the entire database. Many usres of the database system will not be concerned with all of this information. Instead, such usres need only a part of the database. To simplify their interaction with the system , the view level of abstraction is defined.

1.4.3 Data Modeling

Underlying the structure of a database is the concept of a data model, a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. The various data models that have been proposed fall into three groups ; object-based logical models , record-based logical models , and physical data models.

1. Object-based logical models : These models are used in describing data at the conceptual and view levels. They are characterized by the fact they provide fairly flexible structuring capabilities and allow data constraints to be specified explicitly. There are many different models and more are likely to come.
    - The entity-relationship model:This model is based on perception of a real world which consists of a collection of basic objects called entities , and relationships among these objects. An entity is an object that is distinguishable from other objects by a specific set of attributes. A relationship is an association among several entities.
    - The object-oriented model:This model also is based on a collection of objects. An object contains values stored in instance variables within the object. Those values are themselves objects. Thus , objects can contain objects to an arbitrarely deep level of nesting. An object also contains bodies of code that operate on the object. These bodies are called methods. Objects that contain the same types of values and the same methods are grouped together into classes. A class may be viewed as a type definition for objects.

- The binary model:
- The semantic data model.:
- The infological model:
- The functional data model:

2. Record-based Logical Models **:** These are used in describing data at the conceptual and view levels. They are used both to specify the overall logical structure of the database and to provide a higher-level description of the implementation. Record-based models are so named because the database is structured in fixed-format records of several types. Each record type defines a fixed number of fields , or attributes , and each file is usually of a fixed length.

   Record-based models do not include a mechanism for the direct representation of code in the database. Instead , there are seperate languages that are associated within the model to express database queries and updates.

   The three most widely accepted data models are the relational , network and hierarchical models.

   - Relational Model : This represents data and relationships among data by a collection of tables , each of which has a number of columns with unique names.

   - Network Model : Data in the network model are represented by collections of records and relationships among data are represented by links , which can be viewed as pointers.

   - Hierarchical Model : This model is similar to the network model in the sense that data and relationships among data are represented by records and links , respectively. It differs from network model in that the records are organized as collections of trees rather than orbitrary graphs.

3. Physical Data Models : Physical data models are used to describe data at the lowest level. In contrast to logical data models , there are very few physical data models in use. Two of widely known ones are :

    - Unifying Model
    - Frame Model

1.4.4 Instances and Schemes

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment in time is called an instance of the database. The overall design of the database is called the database scheme. Schemes are changed infrequently.

The concept of a database scheme correspondes to the programming language notion of type definition. A variable of a given type has a particular value at a given instant in time. Thus this concept of the value of a given variable in programming language correspondes to the concept of an instance of a database scheme.

Database systems have several schemes , partitioned according to the levels of abstraction. At the lowest level is physical scheme ; at the intermediate level , the concetual scheme ; at the highest level , a subscheme. In general database systems support one physical scheme , one conceptual scheme , and several subschemes.

1.4.4.1 Data Dependence

The ability to modify a scheme definitionin one level without affecting a scheme definition in the next higher-level is called data independence. There are two levels:

1. Physical Data Independence : This is the the ability to modify the physical scheme without causing application programs to be rewritten. Modifications at the physical level are occasionally necessary in order to improve performance.
2. Logical Data Independence : This is the ability to modify the conceptual scheme without causing application programs to be rewritten. Modifications at the conceptual level are necessary whenever the logical structure of the database is altered.

Logical data independence is more difficult to achieve than physical data independence since application programs are heavily dependent on the logical structure of the data they access.

The concept of data independence is similar in many repects to the concept of abstract data types in modern programming languages. Both hide implementation details from the users.

1.4.5 Data Definition Language

A database scheme is specified by a setof definitons which are expressed by a special language called a data definiton language (DDL). The result of compilation of DDL statements is a set of tables which are stored in a special file called dictionary. A data directory is a file that contains metadata ; that is "data about data".

The storage structure and access methods used by the database system are specified by a set of definitions in a special type of DDL called a data storage and definition language. The result of compilation of these definitions is a set of instructions to specify the implementation details of the database schemes which are usually hidden from the users.

1.4.6 Data Manipulation Language

By data manipulation these are meant:

1. The retrieval of information stored in the database .
2. The insertion of information stored in the database.
3. The deletion of information from the database.
4. The modification of data stored in the database.

A data manipulation language (DML) is a language that enables users to access or manipulate data as organizedby the appropriate data model. There are basically two types :

1. Procedural DMLs require a user to specify what data is needed and how to get it.
2. Nonprocedural DMLs require a user to specify what data is needed without specifying how to get it.

Nonprocedural DMLs are usully easier to learn and use than procedural DMLs. However , since a user does not have to specify how to get the data , these languages may generate code which is not as efficient as that produced by procedural languages.

A query is statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a query language. Although technically incorrect , it is common practise to use the terms query language and data manipulation language synonymously.

1.4.7 Database Manager

Databases typically require a large amount of storage space , may be terrabytes of data. Since the main memory of computers cannot store this information , it is stored in disks. Data is moved between disk storage and main memry as needed. Since , the movement of data to and from disk is slow relative to the speed of the CPU , it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory.

The goal of a database system is to simplify and facilitate access to data. A database manager is program module which provides the interface between the low level data stored in the database and the application programs and queries submitted to the system. The database manager is responsible for the following tasks:

Chapter II: Oracle and Java

2.1 Application Structure and Elements

An application is created from classes. It stores related data in *fields*, where the fields can be different types. So we can store a text string in one field, an integer in another field, and a floating point in a third field. A class also defines the *methods* to work on the data.

A very simple class might store a string of text and define one method to set the string and another method to get the string and print it to the console. Methods that work on the data are called *accessor* methods.

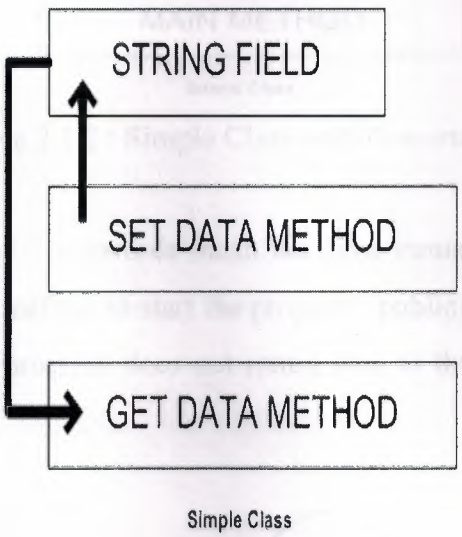

Figure 2.1.1: Simple Class

Every application needs one class with a `main` method. This class is the entry point for the program, and is the class name passed to the `java` interpreter command to run the application.

The code in the `main` method executes first when the program starts, and is the control point from which the controller class accessor methods are called to work on the data.

It has no fields or accessor methods, but because it is the only class in the program, it has a `Main` method.

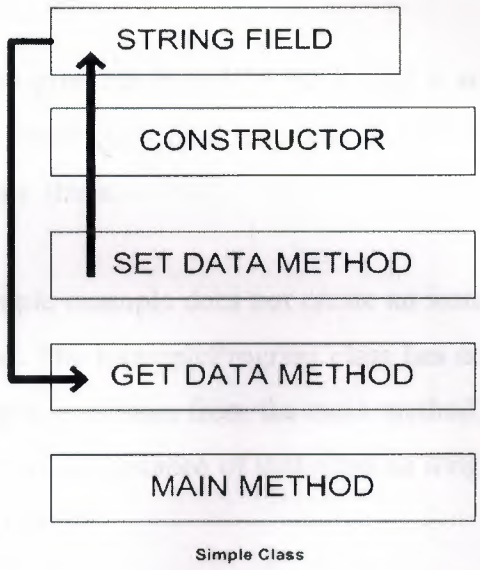

Figure 2.1.2 : Simple Class with Constructor

The `public static void` keywords mean the Java virtual machine (JVM) interpreter can call the program's `main` method to start the program (public) without creating an instance of the class (static), and the program does not return data to the Java VM interpreter (void) when it ends.

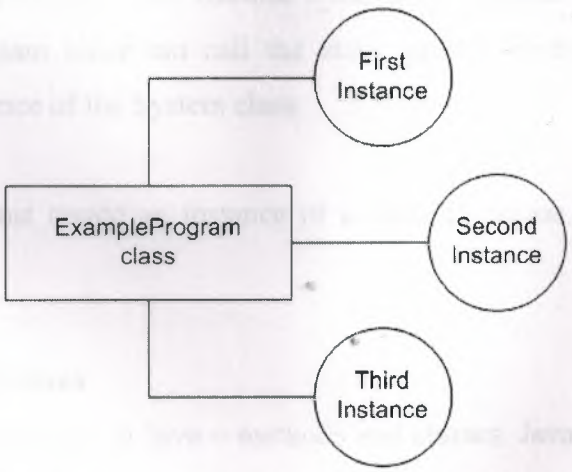

Figure 2.1.3 : A Class with Three Instances

An instance of a class is an executable copy of the class. While the class describes the data and behavior, we need a class instance to acquire and work on data. The diagram at the top

shows three instances of the ExampleProgram class by the names : FirstInstance, SecondInstance and ThirdInstance.

The main method is static to give the Java VM interpreter a way to start the class without creating an instance of the control class first. Instances of the control class are created in the main method after the program starts.

The main method for the simple example does not create an instance of the ExampleProgram class because none is needed. The ExampleProgram class has no other methods or fields, so no class instance is needed to access them from the main method. The Java platform lets us to execute a class without creating an instance of that class as long as its static methods do not call any non-static methods or fields.

The ExampleProgram class just calls println, which is a static method in the System class. The java.lang.System class, among other things, provides functionality to send text to the terminal window where the program was started. It has all static fields and methods.

The static fields and methods of a class can be called by another program without creating an instance of the class. So, just as the Java VM interpreter command could call the static main method in the ExampleProgram class without creating an instance of the ExampleProgram class, the ExampleProgram class can call the static println method in the System class, without creating an instance of the System class.

However, a program must create an instance of a class to access its non-static fields and methods

2.2 Program Modules in Java

There are two kinds of modules in Java – methods and classes. Java programs are written by combining new methods and classes available in the Java Application Programming Interface (Java API or Java class library) and various other class libraries. The Java API provides a rich collection of classes that contain methods for performing common mathematical calculations, character manipulations, input/output operations, error checking and many other useful operations. This set of classes makes writing programs easier.

The Java API classes are part of the Java 2 Software Development Kit (J2SDK) which contais thousands of prepacaged classes.

Methods allow the programmer to modularize a program by separeting its tasks into self-contaned units are somrtimes referred to as programmer-declared methods. The actual statements implementing the methods are written only once anda re hidden from other methods.

There are several motivations for modularizing a program by means of methods. One motivation is that of the divide-and-conquer approach makes program development more mamageable. Another is software reuseability-using existing methods as building blocks to create new programs. Often we can create programs from standardized methods rather than building rdized methods rather than building cutomized code. A third motivation is to avoid repeating code within the program. Packaging code as a method allows a program to execute that code from several locations in a program simply by calling the methods. Also, methods make programs easier to debug and maintain.

A method is invoked or called by a method call. The mehtod call specifies the name of the method and provides information (as arguments) that the called method requires to perform its task. When the method call completes, the method either returns a result to the calling method (or caller) or simply returns control to the calling method. An analogy to this program structure is the hierarchical form of management. A boss (caller) asks a worker (the called method) to perform a task and report back (i.e., return) the result after completing the task. The boss method does not know how the worker method performs its designated tasks. The worker may also call other worker methods, unbeknowst to the boss. This "hiding" of implementation details promotes good software engineering.

2.2.1 Method Declarations

the first line of method declaration is called the *method header*. Follwing the method header, *declarations* and *statements* in braces form the *method body*, which is a block. Variables can be declared in any block, and blocks can be nested. A method cannot be declared inside another method.

The basic format of a method declaration is :

```
Return-value-type method-name(parameter1,parameter2,…,parameterN)
{
      declaration and statements
}
```

The *method-name* is any valid identifier. The *return-value-type* is the type of the result returned by the method to the caller. The *return-value-type* **void** indicates that a method does not return a value. Methods can return at most one value.

The *parameter*s are declared in a comma-separated list in parenthesis that declares each parametre's type and name. There must be one argument in the method call for each parameter in the method declaration. Also, each argument must be compatible with the type of the corresponding parameter. For example, a parameter of type **double** can receive values 7.35,22 or -0,03456 , but not "hello" because a string cannot be implicitly converted to a double variable. If a method does not accept any arguments, the parameter list is empty.
There are three ways to return control to the statement that calls a method. If the method does not return a result, control returns when the program flow reaches the method ending right brace or when the staement **return;** is executed. If the method returns a result, the staement **return *expression;*** evaluates the *expression*, then returns the resulting value to the caller. When a **return** statement executes, control returns immediately to the statement thet called the method.

2.2.2 Constructor Declarations

Constructors are methods that are used to initialize newly created objects of a class. They are declared as follows:

constructorModifiers constructorNameAndParameters throwsClause constructorBody

The constructor modifiers are `public`, `protected`, and `private`. They control access to the constructor and are used in the same manner as they are for variables.

The constructor name is the same as the class name in which it is declared. It is followed by a parameter list, written as follows:

```
(parameterDeclarations)
```

The parameter list consists of an opening parenthesis followed by zero or more parameter declarations followed by a closing parenthesis. The parameter declarations are separated by commas. Parameter declarations are written as follows:

```
type parameterName
```

Each parameter declaration consists of a type followed by a parameter name. A parameter name may be followed by sets of matched brackets (`[]`) to indicate that it is an array.

The `throws` clause identifies all uncaught exceptions that are thrown within the constructor. It is written as follows:

```
throws uncaughtExceptions
```

The exceptions are separated by whitespace characters.

The body of a constructor specifies the manner in which an object of the constructor's class is to be initialized. It is written as follows:

```
{constructorCallStatement blockBody}
```

The *`constructorCallStatement`* and *`blockBody`* are optional, but the opening and closing braces must be supplied.

The constructor call statement allows another constructor of the class or its superclass to be invoked before the constructor's block body. It is written as follows:

```
this(argumentList);
```

```
super(argumentList);
```

The first form results in a constructor for the current class being invoked with the specified arguments. The second form results in the constructor of the class's superclass being invoked. The argument list consists of expressions that evaluate to the allowed values of a particular constructor.

If no constructor call statement is specified, a default `super()` constructor is invoked before the constructor block body.

2.2.3 Statements

2.2.3.1 Empty Statement

The *empty statement* performs no processing. It consists of a single semicolon (`;`).

2.2.3.2 Block Statement

A *block statement* consists of a sequence of statements and local variable declarations that are treated as a single statement block. The statements are enclosed within braces (`{` and `}`).

2.2.3.3 Method Invocation

A *method invocation* invokes a method for an object or a class. Method invocations may be used within an expression or as a separate statement. To be used as a separate statement, the method being invoked must be declared with a void return value. Method invocation statements take the following forms:

```
objectName.methodName(argumentList);

className.methodName(argumentList);
```

The `argumentList` consists of a comma-separated list of zero or more expressions that are consistent with the method's parameters.

2.2.3.4 Allocation Statements

When an object is *allocated*, it is typically assigned to a variable. However, it is not required to be assigned when it is allocated. An allocation statement is of the following form:

```
new constructor(argumentList);
```

The `new` operator is used to allocate an object of the class specified by the constructor. The constructor is then invoked to initialize the object using the arguments specified in the argument list.

2.2.3.5 Statement Labels

A statement can be *labeled* by prefixing an identifier to the statement as follows:

```
label: statement
```

The *label* can be a name or an integer.

2.2.3.6 The `break` Statement

The `break` statement is used to transfer control to a labeled statement or out-of-statement block. It takes the following forms:

```
break;

break label;
```

The first form transfers control to the first statement following the current statement block. The second form transfers control to the statement with the identified label.

2.2.3.7 The Continue Statement

The `continue` statement is used to continue execution of a loop (`for`, `do`, or `while`) without completing execution of the iterated statement. The `continue` statement may take an optional label. It is written as follows:

```
continue label;
```

If a label is supplied, the loop continues at the labeled loop.

2.2.3.8 The `synchronized` Statement

The `synchronized` statement is used to execute a statement after acquiring a lock on an object. It is written as follows:

```
synchronized ( expression ) statement
```

The expression yields the object for which the lock must be acquired.

2.2.3.9 The `try` Statement

The `try` statement executes a block of statements while setting up exception handlers. If an exception occurs the appropriate handler, if any, is executed to handle the exception. A `finally` clause may also be specified to perform absolutely required processing. The `try` statement is written as follows:

```
try block catchClauses finallyClause
```

At least one `catch` clause or a `finally` clause must be provided.

The format of the `catch` clause is as follows:

```
catch (exceptionDeclaration) block
```

If an exception is thrown within the block executed by the `try` statement and it can be assigned to the type of exception declared in the `catch` clause, the block of the `catch` clause is executed.

The `finally` clause, if it is provided, is always executed regardless of whether an exception is generated.

2.2.3.10 The `return` Statement

The `return` statement is used to return an object or a value as the result of a method's invocation. It is written as follows:

```
return expression;
```

The value of the expression must match the return value identified in the method's declaration.

2.3 Application to Applets

2.3.1 The Basic Structure of an Applet

The Java API Applet class provides what is needed to design the appearance and manage the behavior of an applet. This class provides a graphical user interface (GUI) component called a panel and a number of methods. To create an applet, we extend (or subclass) the Applet class and implement the appearance and behavior.

The applet's appearance is created by drawing onto the Panel or by attaching other GUI components such as push buttons, scrollbars, or text areas to the Panel. The applet's behavior is defined by implementing the methods.

An *applet* is simply part of a Web page, like an image or a line of text. Just as a browser takes care of displaying an image referenced in an HTML document, a Java-enabled browser locates and runs an applet. When a Java-capable Web browser loads the HTML document, the Java applet is also loaded and executed. It doesn't matter whether or not the applet is currently available on hard drive. If necessary, the Web browser automatically downloads the applet before running it.

There's a client/server relationship between a browser that wants to display an applet and the system that can supply the applet. The *client* is a computer that requires services from another system; the *server* is the computer that provides those services. In the case of a Java applet, the client is the computer that's trying to display an HTML document that contains a reference to an applet. The server is the computer system that uploads the applet to the client, thereby allowing the client to use the applet.

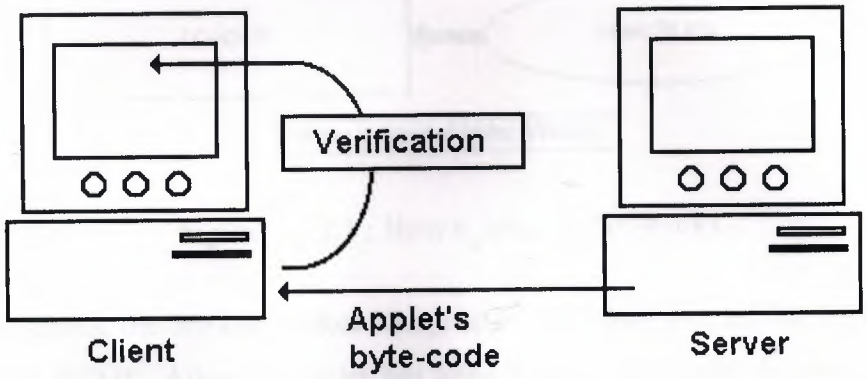

Figure 2.3.1.1 : Verification of Applet's Byte-Code

Java applets are a secure way to transmit programs on the Internet. This is because the Java interpreter will not allow an applet to run until the interpreter has confirmed that the applet's byte-code has not been corrupted or changed in some way (Fig. 2.3.1). Moreover, the interpreter determines whether the byte-code representation of the applet sticks to all of Java's rules. For example, a Java applet can never use a pointer to gain access to portions of computer memory for which it doesn't have access. The bottom line is that, not only are Java applets secure, they are virtually guaranteed not to crash the system.

2.3.2 How Applets Work

Applet technology is the driving force behind the intensity of the Java revolution. Applets are standard Java programs with a few special hooks into the Web browser's environment. The capability of applets to take advantage of the resources provided by a Web browser's environment is what allows them to be easily and powerfully integrated within Web pages.

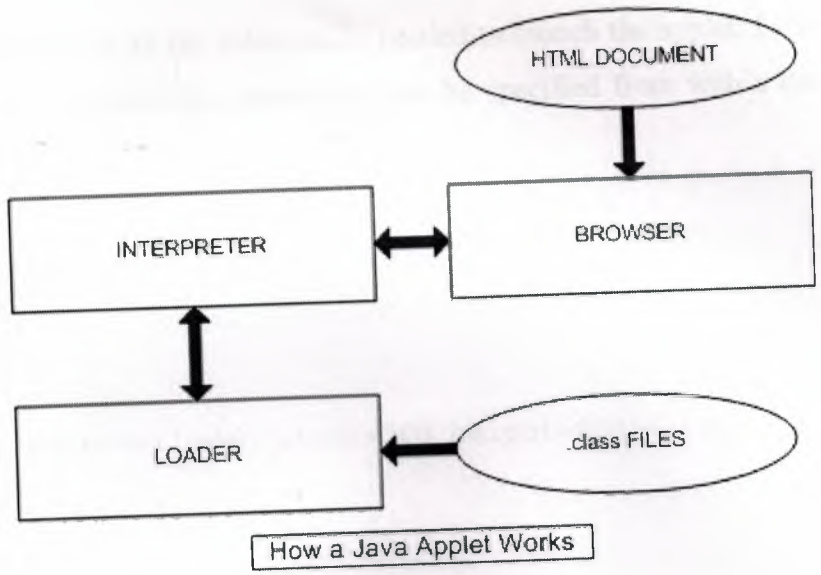

Figure 2.3.2.1 : How a Java Applet Works

As the figure shows, the browser makes a request to the loader to fetch the applet specified in the document's HTML. After the applet has been fetched, the applet begins to execute. The applet is executed by the Java runtime interpreter attached to the browser. The browser acts as a conduit between the Java Virtual Machine inside the interpreter and the outside user interface.

The `Applet` class provides an application framework and tools to access the facilities provided by the browser. Via the browser, the applet has access to graphics, sound, and network capabilities. The `Applet` class can be viewed as merely a wrapper around the capabilities provided by the browser.

2.3.3 The Relationship Between HTML and Applets

An applet is like a child application of the browser. The browser launches the applet in a predefined environment inside the browser. In turn, the browser obtains the information pertaining to the applet's environment from the current document's HTML. In this sense, the relationship between HTML and an applet is that of a command line executing a program.

From within HTML, the syntax to specify the execution of an applet is provided by the applet and parameter tags.

The *applet tag* provides all the information needed to launch the applet. Everything from the base directory to command-line parameters can be specified from within this tag. Here's an example:

```
<html>
<head>
</head>
<body>
    <applet code=HelloWeb width=300 height=200></applet>
</body>
</html>
```

From this HTML command line, the browser is told how to launch the `HelloWeb.class` file. The HTML merely specifies a command line of sorts to the browser.

2.3.4 Applets and Interactive Web Pages

An applet is a Java program designed to run in the environment provided by a Web browser. Inside the browser, an application has the capability to display images, play audio files, and access the Internet. The `Applet` class provides methods to tap these resources provided by the Web browser.

Because of applets are executed locally on our machine, we are able to interact with the applet as part of the Web page's display. Remote processing approach also suffers from the difficulties involved in maintaining information about the state of the applications it supports. The Java model of local execution is able to support a high degree of interactivity. All state information is maintained within the local browser environment and is not distributed between the browser and Web server.

2.3.5 The Execution of an Applet

When the browser comes across the applet tag, it begins gathering the information needed to launch the applet. After the HTML document has been completely interpreted and displayed, the Java runtime interpreter is requested to execute the applet.

When the interpreter receives the request to execute the applet, it executes a loader mechanism to fetch the binary file. After the file is successfully transferred onto the local

machine, it undergoes a number of tests to verify its security and stability. If all is well, the interpreter begins execution of the applet.

Execution continues until the applet terminates or the current browser document is dismissed. This can occur in a couple of different ways: The user might jump to another URL or the browser might terminate. In either case, the applet is terminated.

2.4 Class Declarations

Class declarations allow new classes to be defined for use in Java programs. Classes are declared as follows:

```
classModifiers class className extendsClause implementsClause
classBody
```

The class modifiers, `extends` clause, and `implements` clause are optional. The class modifiers are `abstract`, `public`, and `final`. An `abstract` class provides an abstract class declaration that cannot be instantiated. `Abstract` classes are used as building blocks for the declaration of subclasses. A class that is declared as `public` can be referenced outside its package. If a class is not declared as `public`, it can be referenced only within its package. A `final` class cannot be subclassed. A class cannot be declared as both `final` and `abstract`.

The `extends` clause is used to identify the immediate superclass of a class and thereby position the class within the overall class hierarchy. It is written as follows:

`extends` *`immediateSuperclass`*

The `implements` clause identifies the interfaces that are implemented by a class. It is written as follows:

`implements` *`interfaceNames`*

`interfaceNames` consists of one or more interface names separated by commas.

The class body declares the variables, constructors, and access methods of a class. It is written as follows:

```
{ fieldDeclarations }
```

fieldDeclarations consists of zero or more variable, constructor, or access method declarations or static initializers.

2.4.1 Extending a Class

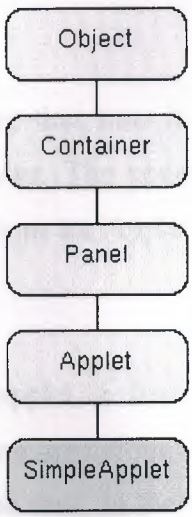

Figure 2.4.1.1 : Extending a Class

Most classes of any complexity extend other classes. To extend another class means to write a new class that can use the fields and methods defined in the class being extended. The class being extended is the parent class, and the class doing the extending is the child class. Another way to say this is the child class inherits the fields and methods of its parent or chain of parents. Child classes either call or override inherited methods. This is called single inheritance. The SimpleApplet class extends Applet class, which extends the Panel class, which extends the Container class. The Container class extends Object, which is the parent of all Java API classes.

The Applet class provides the init, start, stop, destroy, and paint methods. The SimpleApplet class overrides these methods to do what the SimpleApplet class needs them to do. The Applet class provides no functionality for these methods.

However, the Applet class does provide functionality for the setBackground method,which is called in the init method. The call to setBackground is an example of calling a method inherited from a parent class in contrast to overriding a method inherited from a parent class. Java language provides methods without implementations. It is to provide conventions for everyone to use for consistency across Java APIs. If everyone wrote their own method to start an applet, for example, but gave it a different name such as begin or go, the applet code would not be interoperable with other programs and browsers, or portable across multiple platforms. For example, Netscape and Internet Explorer know how to look for the init and start methods.

2.4.2 Behavior

An applet is controlled by the software that runs it. Usually, the underlying software is a browser, but it can also be appletviewer. The underlying software controls the applet by calling the methods the applet inherits from the Applet class.

2.4.2.1 The init Method:

The init method is called when the applet is first created and loaded by the underlying software. This method performs one-time operations the applet needs for its operation such as creating the user interface or setting the font. In the example, the init method initializes the text string and sets the background color.

2.4.2.2 The start Method:

The start method is called when the applet is visited such as when the end user goes to a web page with an applet on it. The example prints a string to the console to tell you the applet is starting. In a more complex applet, the start method would do things required at the start of the applet such as begin animation or play sounds.

After the start method executes, the event thread calls the paint method to draw to the applet's Panel. A thread is a single sequential flow of control within the applet, and every applet can run in multiple threads. Applet drawing methods are always called from a dedicated drawing and event-handling thread.

2.4.2.3 The Stop and Destroy Methods:

The stop method stops the applet when the applet is no longer on the screen such as when the end user goes to another web page. The example prints a string to the console to tell you the applet is stopping. In a more complex applet, this method should do things like stop animation or sounds.

The destroy method is called when the browser exits. Your applet should implement this method to do final cleanup such as stop live threads.

2.5 Packages

The package groups together class libraries, such as the libraries containing information about different commercial properties. A package is the largest logical unit of objects in Java. Packages in Java group a variety of classes and/or interfaces together. In packages, classes can be unique compared with classes in other packages. Packages also provide a method of handling access security. Finally, packages provide a way to "hide" classes, preventing other programs or packages from accessing classes that are for internal use of an application only.

2.5.1 The package Statement

Java programs are organized into *packages*. Packages contain the source code declarations of Java classes and interfaces. Packages are identified by the `package` statement. It is the first statement in a source code file:

```
package packageName;
```

If a `package` statement is omitted, the classes and interfaces declared within the package are put into the default package-the package with no name.

The package name and the `CLASSPATH` are used to find a class. Only one class or interface may be declared as `public` for a given source code file.

2.5.2 The import Statement

The `import` statement is used to reference classes and interfaces that are declared in other packages. There are three forms of the `import` statement:

```
import packageName;

import packageName.className;

import packageName.*;
```

The first form allows classes and interfaces to be referenced using the last component in the package name. The second form allows the identified classes and interfaces to be referenced without specifying the name of their package. The third form allows all classes and interfaces in the specified package to be referenced without specifying the name of their package.

2.6 Action Listening

In addition to implementing the ActionListener interface, we have to add the event listener to the JButton components. An action listener is the SwingUI object because it implements the ActionListener interface

2.7 Event Handling

The actionPerformed method is passed an event object that represents the action event that occurred. Next, it uses an if statement to find out which component had the event, and takes action according to its findings.

2.8 Main Method

The main method creates the top-level frame, sets the title, and includes code that lets the end user close the window using the frame menu.

The code for closing the window shows an easy way to add event handling functionality to a program. If the event listener interface provides more functionality than the program actually uses, use an adapter class. The Java APIs provide adapter classes for all listener interfaces with more than one method. This way, we can use the adapter class instead of the listener interface and implement only the methods we need. In the example, the WindowListener interface has 7 methods and this program needs only the windowClosing method so it makes sense to use the WindowAdapter class instead.

This code extends the WindowAdapter class and overrides the windowClosing method. The new keyword creates an anonymous instance of the extended inner class. It is anonymous because we are not assigning a name to the class and we cannot create another instance of the class without executing the code again. It is an inner class because the extended class definition is nested within the SwingUI class.

This approach takes only a few lines of code, while implementing the WindowListener interface would require 6 empty method implementations

```
WindowListener l = new WindowAdapter() {
 //The instantiation of object l is extended to
 //include this code:
     public void windowClosing(WindowEvent e){
        System.exit(0);
     }
   };
   frame.addWindowListener(l);
```

2.9 File Access by Applications and Applets

2.9.1 File Access by Applications

The Java® 2 Platform software provides a rich range of classes for reading character or byte data into a program, and writing character or byte data out to an external file, storage device, or program. The source or destination might be on the local computer system where the program is running or anywhere on the network.

1. Reading: A program opens an input *stream* on the file and reads the data in serially (in the order it was written to the file).
2. Writing: A program opens an output stream on the file and writes the data out serially.

The conversion from the SwingUI.java program to the FileIO.java program primarily involves the constructor and the actionPerformed method as described.

2.9.2 Exception Handling

An exception is a class that descends from either java.lang.Exception or java.lang.RuntimeException that defines mild error conditions our program might encounter. Rather than letting the program terminate, we can write code to handle exceptions and continue program execution.

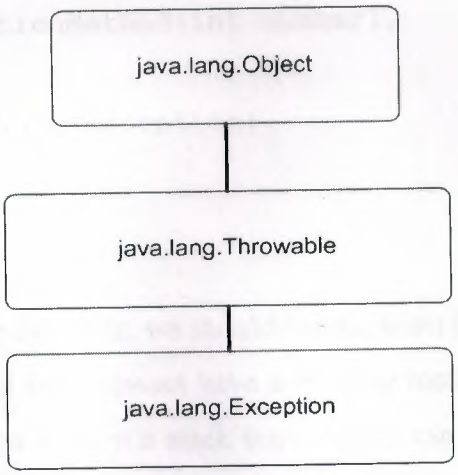

Figure 2.9.2.1 : Exception Handling

The file input and output code in the actionPerformed method is enclosed in a try and catch block to handle the java.lang.IOException that might be thrown by code within the block. java.lang.IOException is what is called a checked exception. The Java platform requires that a method catch or specify all checked exceptions that can be thrown within the scope of a method.

Checked exceptions descend from java.lang.Throwable. If a checked exception is not either caught or specified, the compiler throws an error.

In the example, the try and catch block catches and handles the java.io.IOException checked exception. If a method does not catch a checked exception, the method must specify that it can throw the exception because an exception that can be thrown by a method is really part of the method's public interface. Callers of the method must know about the exceptions that a method can throw so they can take appropriate actions.

However, the actionPerformed method already has a public interface definition that cannot be changed to specify the java.io.IOException, so in this case, the only thing to do is catch and handle the checked exception. Methods that we define can either specify exceptions or catch and handle them, while methods override must catch and handle checked exceptions. Here is an example of a user-defined method that specifies an exception so callers of this method can catch and handle it:

```
public int aComputationMethod(int number1,
      int number2)
      throws IllegalValueException{
    //Body of method
  }
```

When we catch exceptions in our code, we should handle them in a way that is friendly to our end users. The exception and error classes have a toString method to print system error text and a printStackTrace method to print a stack trace, which can be very useful for debugging our application during development.

We can provide our own application-specific error text to print to the command line, or display a dialog box with application-specific error text. Using application-specific error text that we provide will also make it much easier to internationalize the application later on because we will have access to the text.

For the example programs, the error message for the file input and output is handled with application-specific error text that prints at the command line as follows:

```
//Do this during development
  }catch(java.io.IOException e){
   System.out.println(e.toString());
   System.out.println(e.printStackTrace());
  }

//But deploy it like this
  }catch(java.io.IOException e){
   System.out.println("Cannot access text.txt");  }
```

If we want to make our code even more user friendly, we could separate the write and read operations and provide two try and catch blocks. The error text for the read operation could be *Cannot read text.txt*, and the error text for the write operation could be *Cannot write text.txt*.

2.9.3 File Access by Applets

The file access code for the FileIOAppl.java code is equivalent to the FileIO.java application, but shows how to use the APIs for handling data in character streams instead of byte streams. We can use either approach in applets or applications. The choice to handle data in bytes streams in the application and in character streams in the applet is purely random. In real-life programs, we will base the decision on our specific application requirements.

The changes to instance variables and the constructor are identical to the application code, and the changes to the actionPerformed method are nearly identical with these two exceptions:

1. Writing: When the textField text is retrieved, it is passed directly to the out.write call.
2. Reading: A character array is created to store the data read in from the input stream.

2.10 Java and Oracle Security Platform

2.10.1 Java Security Features

The developers of Java and Java-enabled browsers have a powerful set of security features in the Java language, compiler, runtime system, and Web browsers. These security features include security mechanisms that have been specifically designed to eliminate potential security vulnerabilities; other mechanisms, although not intentionally designed for security purposes, encumber both deliberate and inadvertent security threats. The following sections describe these security features.

2.10.1.1 Language Security Features

A number of features have been incorporated into the Java language to make it more reliable and capable. Although these features may not have been driven by security concerns, they still help to minimize security risks. The most important of these features is the removal of all pointer-based operations from the Java language. The absence of pointers eliminates entire classes of security vulnerabilities related to memory browsing, the modification of memory-resident code, and illegal access to security-related objects.

Java's use of strong typing also contributes to security. All objects are associated with a well-defined type and cannot be freely converted from one type to another. Methods cannot be used with classes to which they do not apply. Methods cannot return objects of a type that is incompatible with their return type. Strong typing enforces the Java object-oriented approach and prevents numerous kinds of errors that could lead to security-related malfunctions.

2.10.1.2 Compiler Security Features

The Java compiler also provides features that support security. These features are implemented in the form of compiler checks that prevent errors and undesired actions.

The compiler enforces Java's strong typing by generating compilation errors for statements that violate the language's strong typing rules. It ensures that all methods are appropriate for the objects for which they are invoked.

The compiler checks array operations to make sure that they are valid for the array objects being acted on and that memory overrun errors do not occur. These checks are duplicated and extended by the runtime system.

The compiler checks all class, interface, variable, and method accesses to ensure that the accesses are consistent with the access modifiers used in their declaration. This prevents classes, interfaces, variables, and methods from being used in unintended ways and enforces the information hiding capabilities provided by the access modifiers.

The compiler generates code that treats `String` objects as constants and supports `String` operations through the `StringBuffer` class. This eliminates overrun errors that could cause in-memory modification of data or code.

The compiler also prevents uninitialized variables from being read and constants from being modified. These checks eliminate errors resulting from incorrect variable reading and writing.

2.10.1.3 Runtime Security Mechanisms

The Java runtime system is designed to prevent applets from modifying, deleting, or disclosing your files, accessing in-memory programs and data, and misusing network resources.

This is accomplished by preventing applets from accessing files on your computer, not providing or disallowing services that enable control over other programs, data, or the host operating system, and restricting network connections to the host computer from which an applet is loaded. The specific security mechanisms that implement these controls are discussed in the following subsections.

2.10.1.4 Class Loader Security Checks

Applets are loaded over a network using a class loader. The class loader prevents classes that are loaded from the network from masquerading as or conflicting with classes that are resident on the local file system. This ensures that the security-critical classes of the Java API are not replaced by less trustworthy classes that are loaded over a network.

The class loader separates local and network-loaded classes by placing those classes from a particular network host into a name space that is unique to that host. This approach also keeps network-loaded classes from different hosts from conflicting with each other.

2.10.1.5 The Bytecode Verifier

The security of classes that are loaded over a network is verified using the bytecode verifier. The bytecode verifier checks that the loaded classes are correctly formed and that they do not have the capability to violate type and name space restrictions.

The verifier uses a mini theorem prover to prove that the `.class` file initially satisfies certain security constraints and that when it is executed it will always transition into states in which these security constraints are satisfied. This proof by induction verifies that basic security rules will be enforced throughout the execution of the `.class` file. The verifier proves that no illegal conversion between types can occur, that parameters are correct for the methods and instructions to which they apply, that stack operations do not cause overflows or underflows, that access modifiers are enforced, that no forged pointers can be created, and that register operations do not lead to errors.

2.10.1.6 Memory Management and Control

The memory locations of Java classes and objects are determined at runtime based on the platform hosting the runtime system and the current memory allocation maintained by the operating system.

By performing memory layout decisions at runtime, the potential for inducing errors that cause memory overruns and lead to security malfunctions is greatly reduced. This is because it is very difficult to predict the memory locations at which objects will be stored during code execution. Without this knowledge, complex memory overrun attacks are thwarted.

The Java garbage collector reduces the likelihood that an applet or program may make mistakes in its management of memory resources. Since memory deallocation is automatically handled through the garbage collection process, errors resulting from multiple deallocation of the same memory area or failure to deallocate memory are avoided.

Runtime array bounds checking also reduces the likelihood that errors resulting in illegal memory accesses can occur. By confining array operations to valid array locations, these potential security-related errors are prevented.

2.10.1.7 Security Manager Checks

The Java security manager provides a central decision point for implementing Java security rules. This ensures that security access controls are implemented in a manageable and consistent manner. The `SecurityManager` class of the `java.lang` package may be overridden to implement a custom security policy for standalone Java programs such as those that load applets. A `SecurityManager` object cannot, however, be created, invoked, or accessed by a network-loaded applet. This prevents applets from modifying the security policy implemented by the runtime system's `SecurityManager` object.

The applet security policy implemented by the default `SecurityManager` object varies from one browser to another. Netscape Navigator 2.0 implements a security policy that enforces the following rules for applets that are loaded over a network:

1. Applets cannot create or install a class loader or security manager.
2. Applets cannot create classes in the local class name space.
3. Applets cannot access local packages outside the standard packages of the Java API.
4. Applets cannot access files and directories on the local system in any manner.
5. Applets may establish network connections only to the host system from which they were loaded.
6. Applets cannot create or install a content handler, protocol handler, or socket implementation.

7. Applets cannot read system properties that provide information about a user.
8. Applets cannot modify system properties.
9. Applets cannot run other programs or load dynamic link libraries on the local system.
10. Applets cannot terminate other programs or the runtime system.
11. Applets cannot access threads or thread groups that are outside of their thread group.
12. All windows created by an applet must be clearly labeled as being untrusted.

Chapter III: Oracle Development Siute

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.

3.1.1 Advantages of Net8

Net8 provides the following benefits to users of networked applications.

3.1.1.1 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.

3.1.1.2 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.

3.1.1.3 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.

3.1.1.4 Heterogeneous Networking

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

3.1.1.5 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.

3.2 Net8 Features

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

3.2.1 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.

3.2.2 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.

3.2.2.1 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.

3.2.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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).

3.3 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.

3.3.1 Connect Operations

Net8 supports two types of connect operations.

3.3.1.1 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.

3.3.1.2 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.

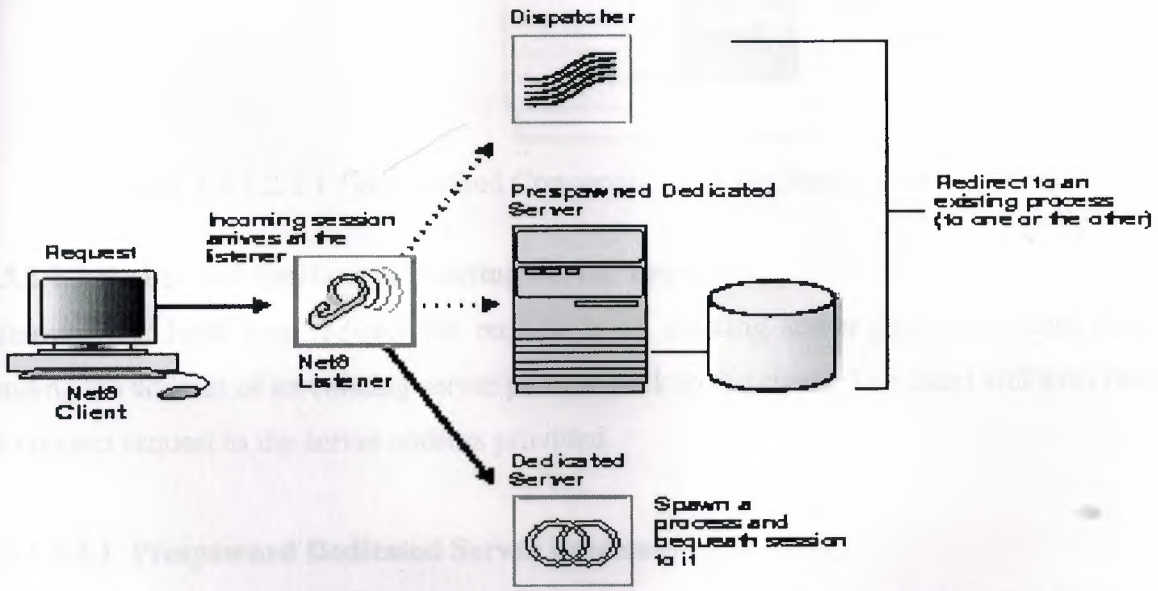

Figure 3.3.1.2.1 : Network Listener In a Typical Net8 Connection

3.3.1.2.1 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.

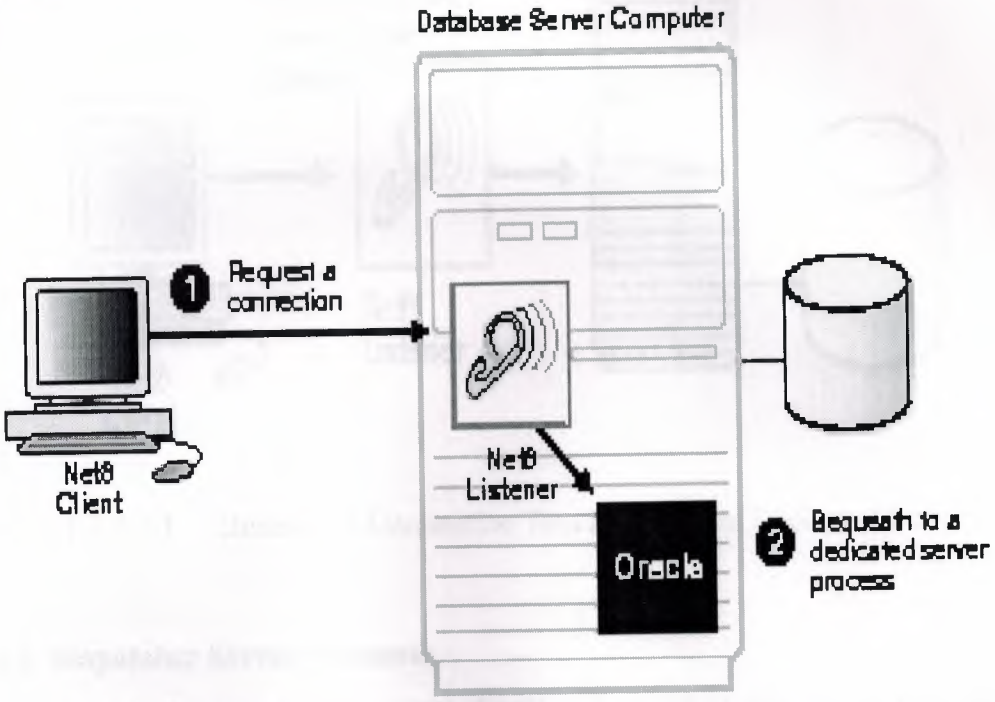

Figure 3.3.1.2.1.1 : Bequeathed Connection To a Dedicated Server Process

3.3.1.2.2 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.

3.3.1.2.2.1 Prespawned 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 prespawned dedicated server processes requires specification in a listener configuration file.

When clients disconnect, the prespawned 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 prespawned server before the timeout expires, the prespawned server shuts down. Figure 3.3.1.2.2.1.1 depicts the role of the network listener in a redirected connection to a prespawned dedicated server process.

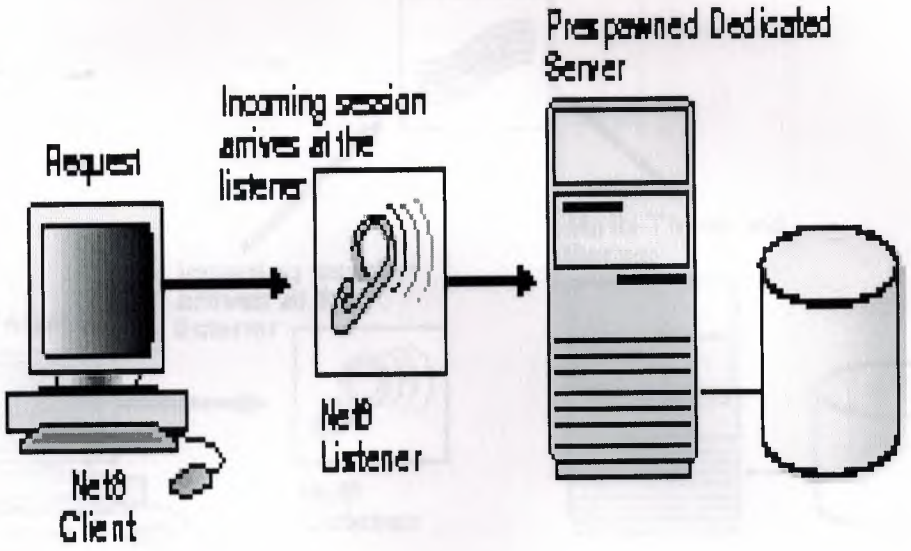

Figure 3.3.1.2.2.1.1 : Redirected Connection To a Prespawned Dedicated Server Process

3.3.1.2.2.2 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 3.3.1.2.2.2.1 depicts the role of the network listener in a redirected connection to a dispatcher server process.

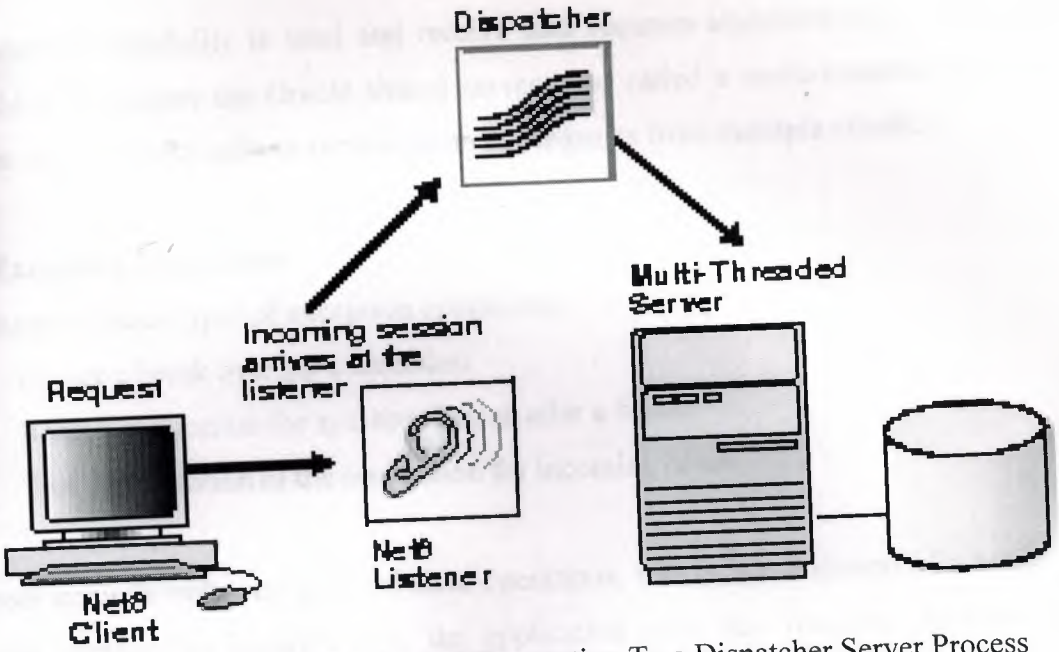

Figure 3.3.1.2.2.2.1 : Redirected Connection To a Dispatcher Server Process

3.3.1.2.3 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.

3.3.2 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.

3.3.3 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.

3.4 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.

3.5 Net8 Architecture

3.5.1 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.

3.5.2 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 3.5.2 depicts a typical OSI Protocol Communications Stack.

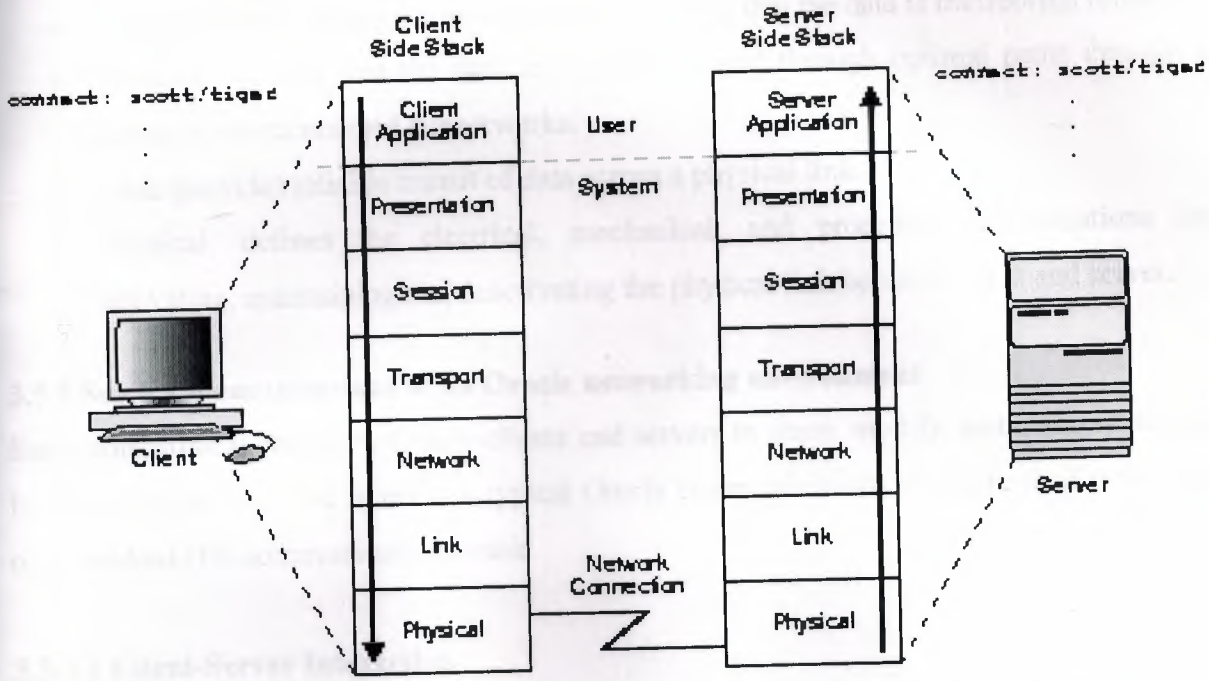

Figure 3.5.2.1 : OSI Communications Stack

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.

3.5.3 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.

3.5.3.1 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.

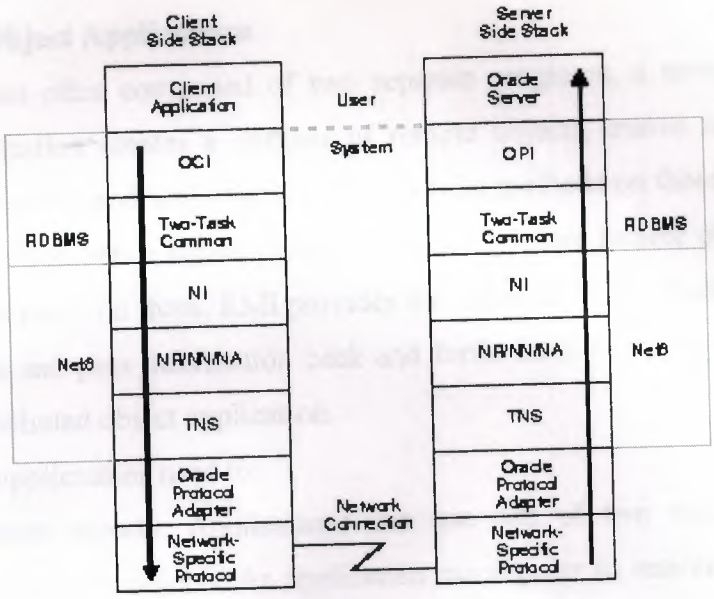

Figure 3.5.3.1.1 : Typical Communications Stack in an Oracle environment

3.6 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 Java™ 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.

3.6.1 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 mecahnism by which the server and the client communicate and pass information back and forth. Such an applications is sometimes referred to as a distributed object application.

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.

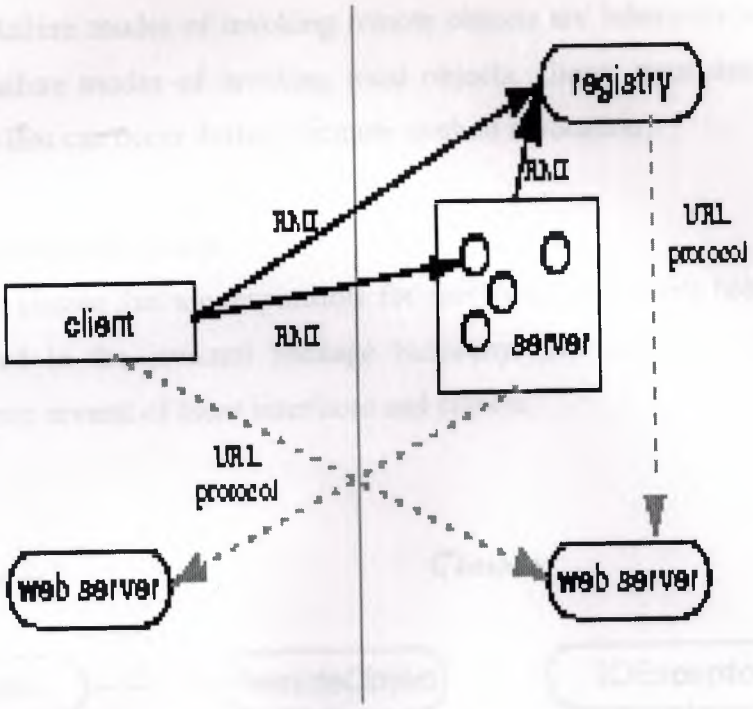

Figure 3.6.1.1 : The Distributed and Nondistributed Models Contrasted

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.

3.6.2 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.

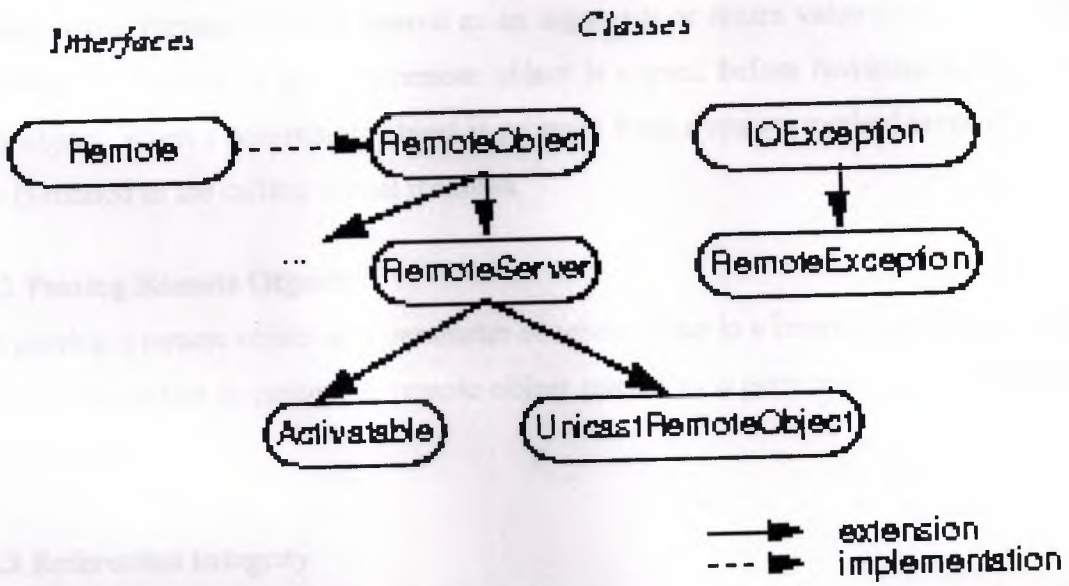

Figure 3.6.2.1 : RMI Interfaces and Classes

3.6.2.1 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.*

3.6.3 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.

3.6.3.1 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.

3.6.3.2 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.

3.6.3.3 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.

3.6.3.4 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.

3.6.3.5 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.

3.6.4 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.

3.6.5 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 reponsible 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.

3.6.6 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.

3.6.7 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.

3.6.8 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.

3.6.9 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.

3.6.9.1 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.

3.6.9.2 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.

6.9.3 Configuring the Client

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

3.6.9.4 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.

3.6.9.5 Performance Issues and Limitations

Calls transmitted via HTTP requests are at least an order of magnitude slower that 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.

onclusions

va's security model is possibly the least understood aspect of the Java system. Because it's usual for a language environment to have security facilities, some people have been thered by the danger; at the same time, because the security restrictions prevent some eful things as well as some harmful things, another group of people has wondered whether curity is really necessary.

ava security is important because it makes exciting new things possible with very little risk. arly security holes caused by implementation bugs are being closed, and technology is being elded that permits the strict security policy to be relaxed carefully and selectively. Resources hat can be used to destroy or steal data are being protected, and researchers are examining ways to prevent applets from using other resources to cause annoyance or inconvenience.

Application developers can design their own security policies and supply parts of the third layer of the Java security model to implement those policies in their applications.The Java security architecture is sound. Early weaknesses and bugs are not a surprise, and the process that has exposed those flaws has also helped remove them.

Java is an interesting new programming language designed to support the safe execution of applets on Web pages. We and others have demonstrated an array of attacks that allow the security of both HotJava and Netscape to be compromised. While many of the specific flaws have been patched, the overall structure of the systems leads us to believe that flaws will continue to be found. The absence of a well-defined, formal security policy prevents the verification of an implementation. We conclude that the Java system in its current form cannot easily be made secure. Significant redesign of the language, the bytecode format, and the runtime system appear to be necessary steps toward building a higher-assurance system. Without a formal basis, statements about a system's security cannot be definitive. The resence of flaws in Java does not imply that competing systems are more secure. We conjecture that if the same level of scrutiny had been applied to competing systems, the results would have been similar. Execution of remotely-loaded code is a relatively new phenomenon, and more work is required to make it safe.

References

Database system concepts
Henry F. Korth
Abraham Silberschatz
McGraw-Hill International Editions
Computer Science Series
Second Edition

Dr. Shuguang Hong, CIS, GSU
DB Driven Web Applications
Database Administration
Database Management Systems
Relational Algebra & SQL

David Toman
University of Waterloo
Database Management Systems
Computer Science 265 Lecture Notes

Drew Dean, Edward W. Felten, Dan S. Wallach
Department of Computer Science
Princeton University
Symposium on Security and Privacy, Oakland, CA, May 6–8, 1996.

S. R. Ames, Jr., M. Gasser, and R. G. Schell
Security kernel design and implementation: An introduction. Computer, July 1983

J. P. Anderson
Computer security technology planning study
Technical Report ESD-TR-73-51, U.S. Air Force,
Electronic Systems Division, Deputy for Command and
Management Systems, HQ Electronic Systems Division
(AFSC), L. G. Hanscom Field, Bedford, MA 01730 USA

N. S. Borenstein
In IFIP International Working Conference on Upper Layer Protocols, Architectures and Applications, 1994.

G. Castagna
Covariance and contravariance: Con-flict without a cause. Technical Report LIENS-94-18, D´epartement de Math´ematiques et d'Informatique, Ecole Normale Sup´erieure, Oct. 1994

W. R. Cheswick and S. M. Bellovin
Firewalls and Internet Security: Repelling theWily Hacker. Addison-Wesley, 1994.

D. Flanagan
Java in a Nutshell. O'Reilly & Associates, Inc., 1st edition, Feb. 1996.

J. Gosling and H. McGilton
The Java Language Environment. Sun Microsystems Computer Company,
http://java.sun.com/whitePaper/

R. Milner, M. Tofte, and R. Harper
The Definition of Standard ML. MIT Press, Cambridge, MA, 1990

National Computer Security Center
Department of Defense Trusted Computer System Evaluation Criteria . National Computer Security Center

J. Roskind

Java and security. In Netscape Internet Developer Conference, Netscape Communications Corp

David Hopwood
Network Security
www.users.zetnet.co.uk/hopwood/papers/compsec97.html

Gary McGraw and Edward Felten
Mobile Code and Security: Why Java Security Is Important

Joseph A. Bank
"Java Security", Dec. 8, 1995
http://swissnet.ai.mit.edu/~jbank/javapaper/javapaper.html

Mike Fletcher
Java 1.1 Unleashed
Java Security
Macmillan Computer Publishing

Sun Microsystems
"Frequently Asked Questions—Applet Security", Jan. 9, 1996 version 1.0 Beta 2
http://java.sun.com/sfaq/
"HotJava: The Security Story", May 19, 1995
http://java.sun.com/1.0alpha3/doc/security/security.html
"The Java Language Specification", DRAFT—Version 1.0 Beta, October 30, 1995
http://java.sun.com/JDK-beta2/psfiles/javaspec.ps

Princeton University
Department of Computer Science
Princeton Secure Internet Programming Team
http://www.cs.princeton.edu/~ddean/java/
http://www.cs.princeton.edu/sip/java-faq.html

ppendices

ppendix A: Program Example

```
port java.sql.*;
port javax.swing.*;
port java.awt.*;
port java.awt.event.*;
port java.util.*;

blic class Test1 extends JApplet implements ActionListener{

    private Connection connection;
    private JTable table;

    JButton btn1;
    JTextField text1;
    JLabel label1;
    JPanel panel1;

    String query="Select * from emp";
    Statement statement;
    ResultSet resultset;

    public void init(){

    panel1=new JPanel();
    panel1.setLayout(new GridLayout(1,3));

    label1=new JLabel("Enter Index here >>");
    panel1.add(label1);

    text1=new JTextField(15);
    panel1.add(text1);

    btn1=new JButton("Sort!");
    btn1.addActionListener(this);
    panel1.add(btn1);

    getContentPane().add(panel1,BorderLayout.NORTH);

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

    connection=DriverManager.getConnection(url,user,passw);
              JOptionPane.showMessageDialog(null,"Connection
Successful");
```

```
            }
            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());
            }

      }


public void paint(Graphics g){

      try{

      statement=connection.createStatement();
      resultset=statement.executeQuery(query);
      displayResultSet(resultset);
      validate();
      resultset.close();
      statement.close();

      }
      catch(SQLException sqlex){
            sqlex.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");
            return;
      }

      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);

    }
    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 destroy(){

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

    }

public void actionPerformed(ActionEvent e){
    if (e.getSource()==btn1){
        query="select * from emp order by "+text1.getText()+"";
        repaint();
    }
    }
}
```