Advanced Java Programming with Database Application

Course Designer and Acquisition Editor

Centre for Information Technology and Engineering

Manonmaniam Sundaranar University

Tirunelveli

Advanced Java Programming with Database Application

CONTENTS

Lecture 1

Database Management Systems

Objectives

In this Lecture will learn the following:

- ❧ What is Database?

- ❧ Database Approach

- ❧ DBMS Functions

- ❧ DBMS Standardization

- ❧ Conceptual data modeling

- ❧ Relational Model

- ❧ CODD'S Rules

Coverage Plan

Lecture 1

1.1 Snap shot

Every organization has a pool of resources that it must manage effectively to achieve its objectives. Although their rule differs all resources human, financial and material share a common characteristic.

The organization that fails to treat data or information as resource and to manage it effectively will be handicapped in how it manages its, manpower, material and financial resources. In order to satisfy the information requirements of management, the data should be stored in an organized form.

What is a Database?

A brief definition might be: THE INFORMATION, HELD OVER A PERIOD OF TIME, IN COMPUTER - READABLE FORM.

Typical examples of information stored for some practical purpose are: Information collected for the sake of making a statistical analysis, e.g. the national census. Operational and administrative information required for running an organization or a commercial concern this will take the form of stock records, personnel records, customer records . . . etc.

Held over a Period of Time

Because of the investment involved in setting up a database, the expectation must be that it will continue to be useful, over years rather than months. But the relationship with time varies from one type of information to another An organizational database may not change very drastically in size, but it will be subject to frequent updating (deletions, amendments, insertions) following relevant actions within the organization itself. Ensuring the accuracy, efficiency and security of this process is the main concern of many database designers and administrators.

The function of the DBMS is to store and retrieve information as required by applications programs or users.

Data verses Information

Data are facts concerning people, places, events or other objects or concepts. Data are often relatively useless to decision-makers until they have been processed or refined in some manner. Information is data that have been processed and refined and then given in the format that is convenient for decision making or other organizational activities. For example, report about student fee paid details is useful information for finance section. Actually data are the facts stored in the record of a database. But the processed facts are presented in a form for usage is information.

Data Base concepts

A database is a shared collection of interrelated data designed to meet the varied information needs of an organization. Consider an example, in a payroll application each person's record has NAME, AGE, DESIGNATION, BASIC PAY . . . etc as columns. So payroll database has collection of all employees of all employees records, that are interrelated. From the database, various reports like payslip, persons with particular designation, service report etc can be obtained. The database acts as a media to store the data in an organized way so that it can be managed effectively. A database has two important properties: It is integrated and shared.

Benefits of the Database approach

The data base approach offers a number of important advantages compared to traditional approach. These benefits include minimal data redundancy, consistency of data, integration of data, sharing of data, enforcement of standards, ease of application development, uniform security, privacy and integrity controls, data accessibility and responsiveness, data independence, and reduced program maintenance.

Minimum data redundancy

With the data base approach, previously separate data files are integrated into a single, logical structure. So each item is ideally recorded in only one place in the database. Hence in a data base system, the data redundancy is controlled.

Consistency of data

By eliminating data redundancy, the consistency of data has greatly improved. If any change in the data, it can be incorporated in one place than the traditional file system.

Integration of data

In a database, data are organized into a single, logical structure, with logical relationships defined between associated data entities.

Sharing of data

The database is intended to be shared by all authorized users in the organization. Since all the data are integrated,

1.2 Summary of DBMS Functions

Data definition

Data can be defined as FILES to RECORD STRUCTURES FIELD NAMES, TYPES and SIZES RELATIONSHIPS between records of different types Extra information to make searching efficient, e.g. INDEXES.

Data entry and validation

Validation may include: TYPE CHECKING, RANGE CHECKING, CONSISTENCY CHECKING

In an interactive data entry system, errors should be detected immediately - some can be prevented altogether by keyboard monitoring - and recovery and re-entry permitted. If the database is error bound then it will be the main cause to make the program error prone.

Updating: Updating of data is very important otherwise it will be a waste in a long run.

Updating involves: Record INSERTION, Record MODIFICATION, Record DELETION.

Updating may take place interactively, or by submission of a file of transaction records; handling these may require a program of some kind to be written, either in a conventional programming language or in a language supplied by the DBMS for constructing command files.

Data retrieval on the basis of selection criteria

For this purpose most systems provide a QUERY LANGUAGE with which the characteristics of the required records may be specified. Query languages differ enormously in power and sophistication but a standard, which is becoming increasingly common, is based on the so-called RELATIONAL operations.

These allow

Selection of records on the basis of particular field values. Selection of particular fields from records to be displayed. Linking together records from two different files on the basis of matching field values. Arbitrary combinations of these operators on the files making up a database can answer a very large number of queries without requiring users to go into one record at a time processing.

Report definition

Most systems provide facilities for describing how summary reports from the database are to be created and laid out on paper. These may include obtaining:

COUNTS, TOTALS, AVERAGES, MAXIMUM and MINIMUM values

On a table over particular CONTROL FIELD above layouts have to be found out. Also specification of PAGE and LINE LAYOUT, HEADINGS, PAGE-NUMBERING, and other narrative to make the report comprehensible.

Security

This has several aspects: Ensuring that only those authorized can see and modify the data, generally by some extension of the password principle.

Ensuring the consistency of the database where many users are accessing and up-dating it simultaneously.

Ensuring the existence and INTEGRITY of the database after hardware or software failure. At the very least this involves making provision for back-up and re-loading.

Why have databases (and a DBMS)?

An organization uses a computer to store and process information because it hopes for speed, accuracy, efficiency, economy etc. beyond what could be achieved using clerical methods. The objectives of using a DBMS must in essence be the same although the justifications may be more indirect.

DBMS Standardization

Early computer applications were based on existing clerical methods and stored information was partitioned in much the same way as manual files. But the computer's processing speed gave a potential for RELATING data from different sources to produce valuable management information, provided that some standardization could be imposed over departmental boundaries. The idea emerged of the integrated database as a central resource. Data is captured as close as possible to its point of origin and transmitted to the database, then extracted by anyone within the organization who requires it. However many provisos have become attached to this idea in practice, it still provides possibly the strongest motivation for the introduction of a DBMS in large organizations. The idea is that any piece of information is entered and stored just once, eliminating duplications of effort and the possibility of inconsistency between different departmental records. Data redundancy has to be removed at the most possible.

Advantages

Organizational requirements change over time, and applications programs laboriously developed need to be periodically adjusted. A DBMS gives some protection against change by taking care of basic storage and retrieval functions in a standard way, leaving the applications developer to concentrate on specific organizational requirements. Changes in one of these areas need not affect elsewhere. In general a DBMS is a substantial piece of software, the result of many man-years of effort. Provide more facilities than would be economic in a one-off product.

The points discussed above are probably most relevant to the larger organization using a DBMS for its administrative functions - the environment in which the idea of databases first originated. In other words the convenience of a DBMS may be the primary consideration. The purchaser of a small business computer needs all the software to run it in package form, written so that the minimum of expertise is required to use it. The same applies to departments (e.g. Research & Development) with special needs that cannot be satisfied by a large centralized system. When comparing database management systems it is obvious that some are designed in the expectation that professional staff will be available to run them, while others are aimed at the total novice.

Actual monetary costs vary widely from, for instance, a large multi-user Oracle system to a small PC-based filing system.

1.3 Data base Project Development

The conventional SYSTEMS LIFE CYCLE of project Development consists of: Analysis, Design, Development, Implementation, Maintenance.

Analysis

A CONCEPTUAL DATA MODEL describing the information which is used inside the organization but not in computer-related terms. The conceptual data model provides a context within which more detailed design specifications can be produced, and should help in maintaining consistency from one application area to another. A CONCEPTUAL PROCESS MODEL describing the functions of the organization in terms of events (e.g. a purchase, a payment, a booking) and the processes which must be performed within the organization to handle them. This may lead to a more detailed functional specification - describing the organizational requirements which must be satisfied, but not how they are to be achieved.

Design

A LOGICAL DATA MODEL is a description of the data to be stored in the database, using the conventions prescribed by the particular DBMS to be used. This is sometimes referred to as a SCHEMA and some DBMS also give facilities for defining SUB-SCHEMA or partitions of the overall schema. A SYSTEM SPECIFICATION, describing in some detail what the proposed system should do. This will now refer to COMPUTER PROCESSES, but probably in terms of INPUT and OUTPUT MESSAGES rather than internal logic. By the end it should give the outline, how the DBMS should be

Development

This is the actual coding phase. Specification of the database itself must now come down another level, to decisions about PHYSICAL DATA STORAGE in particular files on particular devices, etc. Conventional program development - coding, testing, debugging etc. may also be done. It will simply be a matter of discovering how to use the command and query language already supplied to store and retrieve data, generate reports and other outputs. Even here an element of testing and

debugging may be involved, since it is unlikely that the new user of a system will get it exactly right the first time.

Implementation

This puts the work of the previous three phases into everyday use of the endures. It involves such things as loading the database with live rather than test data, staff training, probably the introduction of new working practices.

Maintenance

Systems once implemented generally require further work done on them as time goes by, either to correct original design faults or to accommodate changes in user requirements or in the system level. One of the objectives of using a DBMS is to reduce the impact of such changes - for example the data can be physically re-arranged without affecting the logic of the programs which use it. Some DBMS provide utility programs to re-organize the data when either its physical or logical design must be altered.

Conceptual Data modeling

CONCEPTUAL DATA MODELLING is the first stage in the process of TOP-DOWN DATABASE DESIGN. The aim is to describe the information used by an organization in a way, which is not governed by implementation-level issues and details. It should make it easy to see the overall picture so that non-technical staff can contribute to discussions.

A common method of analysis involves identifying:

ENTITIES (persons, places, things etc.) which the organization has to deal with ATTRIBUTES - the items of information which characterize and describe these entities RELATIONSHIPS between entities which exist and must be taken into account when processing information

In the abstract, or illustrated by superficial examples, this looks a very simple idea. An explanation may put forward a car as a typical entity, point to make, colour, registration number as obvious attributes, and suggest owning and driving as relationships in which it takes part. But applying the method of analysis to some useful purpose in a working organization will be difficult, simply because the world does not fit so neatly into boxes.

Entity

It is any object or event about which the organization chooses to collect and store data. Any entity may be tangible object, such as an employee, a product, a computer, or a customer, or it may be a intangible item, such as bank account, a part failure of a flight.

Attribute

An attribute is a property of an entity that we choose to record.

Enrol No.	Name	Course	Mark	Result

Attribute 1   Attribute 2
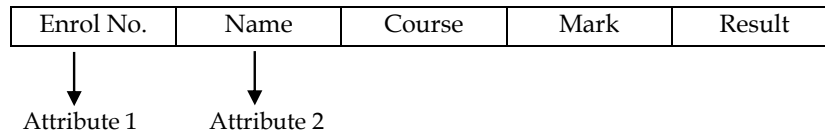
In other words, an entity or a record is nothing but collection of attributes or fields.

Key

A key is a data item used to identify a record. There are two basic types of keys namely Primary and Secondary keys.

A Primary key is a data item that uniquely identifies a record. For example, in the student record, ENRL_NO is a primary key that uniquely identify a student record. Each student record has different ENRL_NO that is no two students are having the same ENRL_NO.

A Secondary Key is a data item that normally does not uniquely identify a record but identifies a group or records in a set that share the same property. For example, in the student record, COURSE is a secondary key that identifies a set of record, having same course.

Enrol No.	Name	Course	Mark	Result

Primary Key   Secondary Key
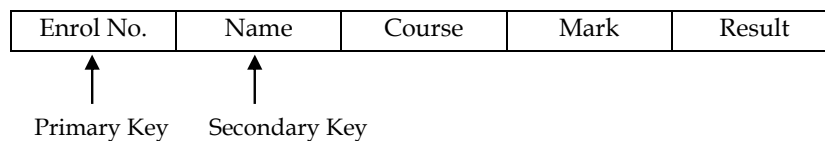
1.4 Types of relationship between data item

A database is nothing b8ut collection of records and a record is a collection of fields or data item and fields is an attribute about the record. When collection of data items are involved, there exists a relationship. If the database contains N data items, then there exist N (N-1) possible associations among the data items.

The various types of association that exists among the data items are

- One to one association
- One to many association
- Many to many association

One to One Association

It means that, at a given instant of time, each value of data item A is associated with exactly one value of data item A is associated with exactly one value of data item B, conversely each value of B is associated with one value of A. For example each student has a name in the student record.

Students   Record Name

A   B
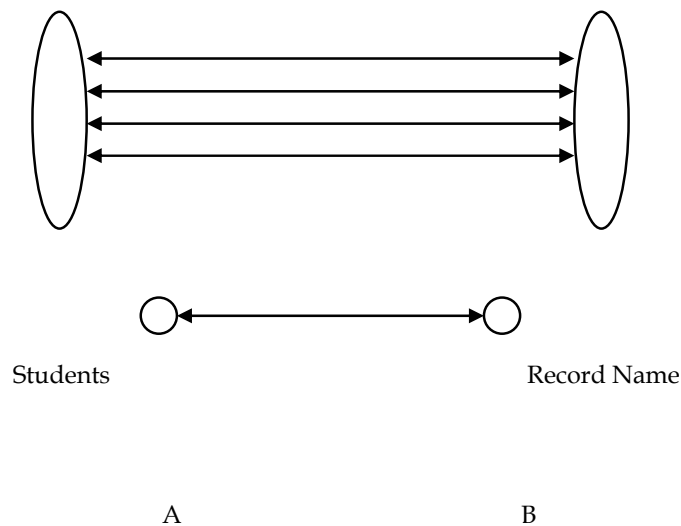
Each Student is assigned with one Name and in reverse each Name is assigned to one Student, so one to one association exists between Students and Names.

One to Many Association

One to many association is, at a given instant of time, each value of data item A is associated with zero, one or more than one value of data item B. However, each value of B is associated with exactly one value of A. The mapping from B to A is said to be many to one, since there may be many values of B associated with one value of A.

Course   Students

A   B
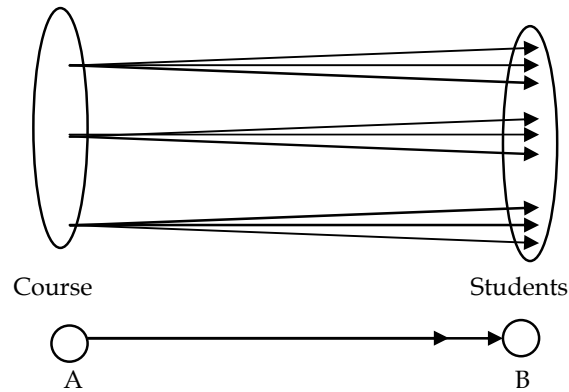
The course and students association is one to many. Since in a course more than one student is engaged.

Many to Many Association

Many to many association means that, at a given instant of time, each value of data item A is associated with zero, one, or many values of data item B. Also each value of B is associated with zero, one or many values of A.

Student   Teacher

A   B
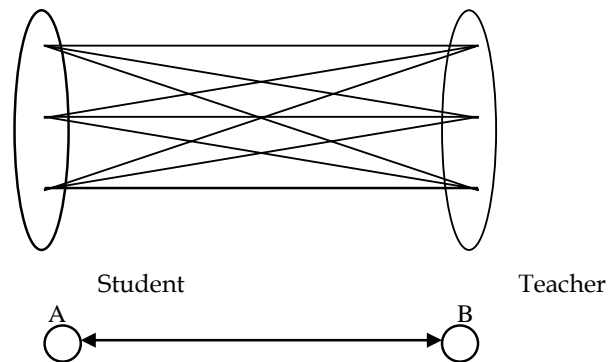
The Student Teacher association is many to many. Since a Student can have more than one Teacher, in the same way a Teacher can have more than one Student.

Relationships

In many applications one external event or process may affect several related entities, requiring the setting of LINKS from one part of the database to another. Important information to be recorded is:

The relational model

The relational model consists of three components:

1. A Structural component -- a set of TABLES (also called RELATIONS).
2. MANIPULATIVE component consisting of a set of high-level operations which act upon and produce whole tables.
3. A SET OF RULES for maintaining the INTEGRITY of the database.

1.5 Codd's Rules

As the benefits of the relational approach become more widely perceived, vendors of DBMSs increasingly often claim that their products are 'relational'. In 1985 Codd produced the following set of 'rules' by which systems should be judged:

Information

Data retrieved from the database should be informative. All information represented in each column should be of specific type.

Guaranteed Access

Each datum (atomic value) in a relational database is guaranteed to be logically accessible through a combination of table-name, primary key value and column-name.

Systematic Treatment of Null Values

Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in a fully relational DBMS. Null values are of two types for representing missing information and inapplicable information.

Database Description

All information stored in the form of table. It is stored in a table, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

Comprehensive Sub-Language

A relational system may support several languages. However there must be at least one language (SQL) and that is comprehensive in supporting all the following items:

1. Data definition
2. View definition
3. Data manipulation (interactive and by program)
4. Integrity constraints
5. Authorisation
6. Transaction boundaries (begin, commit and rollback)

View Updating

All theoretically updateable views are also updateable by the system. Using the structured query language one should be able to Add, delete and modify.

Insert and Update

The capability of handling a base table or a derived table as a single operand applies not only to retrieval of data but also to insertion, updating, and deletion. (This allows the system to optimise its execution sequence by determining the best access paths.)

Physical Data Independence

Application programs and terminal activities, there is no need to specify the physical location of the database. That is path should not be mentioned. (There must be a clear distinction between logical and physical design levels.)

Logical Data Independence

This rule permits logical database design to be changed dynamically, e.g. by splitting or joining base tables in ways which do not entail loss of information. During the retrieval we can change the field orders.

Integrity Independence

Integrity constraints must be definable in the relational data and storable in the catalogue, not in the applications program. Certain integrity constraints hold for every relational database, further application-specific rules may be added. The general rules relate to:

1. Entity integrity: no component of a primary key may have a null value.

2. Referential integrity: for each distinct non-null 'foreign key' value in the database, there must exist a matching primary key value from the same domain.

Distribution Independence

A relational DBMS has distributional independence - i.e. if a distributed database is used it must be possible to execute all relational operations upon it without knowing the physical locations of data. This must apply both when distribution is originally introduced, and when data is redistributed.

Non-Subversion

A low-level (single record-at-a-time) language cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher-level (multiple record-at-a-time) language.

1.6 Short Summary

- Data are real facts.

- Information is processed data….

- The data base approach offers a number of important advantages compared to traditional approach.

- The various types of association that exists among the data items are

  I. One to one association
  II. One to many association
  III. Many to many association

- A database is a shared collection of interrelated data designed to meet the varied information needs of an organization.

1.7 Review Questions

1. Explain the following

   - Database concepts
   - DBMS functions
   - One to One association
   - One to Many association
   - Many to Many association
   - Many to Many association
   - Codd's rules

ဆ...ၶ

Lecture 2

Structured Query Language

Objectives

In this lecture you will learn the following

- Relational and Logical and Comparison Test
- Unions and Joins
- Parent/Child Relationships
- Primary and Foreign Keys
- Create, Alter and Drop Statements
- Insert, Delete and Update Statements

Coverage Plan

Lecture 2

2.1 Snap Shot - Structured Query Language

The JDBC requires JDBC-compliant drivers to support the American National Standards Institute (ANSI) SQL-92 Entry Level version of the standard that was adopted in 1992. SQL is mainly classified into Data Definition Language (DDL) and Data Manipulation Language (DML).

2.2 Using SQL as a Data Definition Language

Using SQL to define a database means creating a database, creating tables and adding them to a database, updating the design of existing tables, and removing tables from a database.

The Create Database Statement. The CREATE DATABASE Statement can be used to create a database:

The CREATE DATABASE DatabaseName.

Substitute the name of the table to be created for databaseName. For example, the following statement created a database named Salesreps

Create Database Salesreps

The Create Table Statement : The CREATE TABLE statement creates a table and adds it to the database:

CREATE TABLE tableName (columnDefinition,… , columnDefinition)

Each columnDefinition is of the form ColumnName columnType

The columnName is unique to a particular column in the table. The columnType identifies the type of data that may be contained in the table. Common data types are

Char(n)	-	An n character text string
Int	-	An integer value
Float	-	A floating point value
Bit	-	A boolean (1 or 0) value

Date   -   A date value

Time   -   A time value

Note: The Types class of java.sql identifies the SQL data types supported by Java. The get methods of the ResultSet interfaces are used to convert SQL datatypes into Java data types. The set methods of the PreparedStatement interface are used to convert Java types into SQL data types. These classes are covered in more detailed in the next chapter.

The following is an example of a CREATE TABLE statement:

```
CREATE TABLE Customers (
CustName char(30),
Company char(50),
Cust_rep integer,
Credit_limit money
)
```

The preceding statement creates a customers table with the following columns:

CustName   -   A 30-character-wide text field

Company   -   A 20-character-wide text field

Cust_rep   -   A integer type

Credit_limit -   A money data type to represent currency values.

The ALTER TABLE Statement

The ALTER TABLE statement adds a row to an existing table or to change the table definitions.

```
ALTER TABLE tableName ADD (columnDefinition… columnDefinition)
```

The row values of the newly added columns are set to NULL. Columns are defined as described in the previous section.

The following is an example of the ALTER TABLE statement that adds a column named Fax to the Contacts table:

ALTER TABLE CUSTOMERS ADD (CONTACT_NAME VARCHAR(20))

The DROP TABLE Statement

The DROP TABLE statement deletes a table from the database:

DROP TABLE tableName

The dropped table is permanently removed from the database. The following is an example of the DROP TABLE statement:

DROP TABLE CUSTOMERS

The preceding statement removes the CUSTOMERS table from the database. Tables once dropped cannot be retrived.

2.3 Using SQL as a Data Manipulation Language

One of the primary uses of SQL is to update the data contained in a database. There are SQL statements for inserting new rows into a database, deleting rows from a database, and updating existing rows.

The INSERT Statement

The INSERT statement inserts a row into a table:

```
INSERT INTO  tableName VALUES ('values 1 ' , …, 'valuen' )
```

In the preceding form of the INSERT statement, value 1 through value n identify all column values of a row. Values should be surrounded by single quotes.

The following is an of the preceding form of the INSERT statement:

```
INSERT INTO SALESREPS VALUES (
1008,
'Joy Anand',
29,
'23',
'Executive',
```

```
105,
500000,
420000
)
```

The preceding statement adds a row to the SALESREP table. All columns of this row are filled in.

An alternative form of the INSERT statement may be used to insert a partial row into a table. The following is an example of this alternative form of the INSERT statement:

```
INSERT      INTO      tableName      (columnName1,….,columnNamem)      VALUE
('value1',…,'valuem')
```

An alternative form of the INSERT statement may be used to insert a partial row into a table. The following is an example of this alternative form of the INSERT statement:

```
INSERT      INTO      tableName      (columnName1,….,columnNamem)      VALUE
('value1',…,'valuem')
```

The values of columnName1 through columnNamem are set to value 1 through valuem. The value of the other columns of a row are set to NULL.

An example of this form of the INSERT statement follows:

```
INSERT INTO ORDERS (ORDER_DATE, PRODUCT)
VALUES
(
'4-DEC-1999',
'Microprocessor'
)
```

The preceding statement adds a order with the date 4th December 1999 and the product Microprocessor to the Order table. The other columns of the table are null.

The DELETE Statement

The DELETE statement deletes a row from a table:

```
DELETE FROM tableName [WHERE condition]
```

All rows of the table that meet the condition of the WHERE clause are deleted from the table. The WHERE clause is covered in a subsequent section of this chapter.

Warning

If the WHERE clause is omitted, all rows of the table are deleted.

The following is an example of the DELETE statement:

```
DELETE FROM orders
WHERE Order_Date = '14-FEB-2000'
```

The preceding statement deletes all ORDERS on 14th February 2000 from the Orders table.

The UPDATE Statement

The UPDATE Statement is used to update an existing row of a table:

UPDATE tableName SET columnName1 = 'value1", …...,columnName = 'value'
[WHERE condition]

All the rows of the table that satisfy the condition of the WHERE clause are updated by setting the value of the specified values. If the WHERE clause is omitted, all rows of the table are updated.

An example of the UPDATE statement follows:

```
UPDATE Customers
SET Credit_Limit = 1200000
WHERE Company = 'Axes Technologies'
```

The preceding statement changes the Credit Limit of the company Axes Technologies with the new Credit Limit of Rs12,00,000.

2.4 Using SQL as a Dataquery Language

The most important use of SQL for many users is for retrieving data contained in a database. The SELECT statement specifies a database query:

SELECT columnList1 FROM table1 ,…, tablem [WHERE condition]

Note: An asterisk (*) may replace columnList1 to indicate that all columns of the table(s) are to be returned.

Example
```
SELECT CITY, SALES FROM OFFICES
SELECT * FROM CUSTOMERS
```

Some relational and logical comparisons can also be included. They are classified mainly into 6 categories. They are

Comparison Test

Comparison test makes use of the relational operators. They are =, <>, <, <=, >, >= used to compare equal to, not equal to, less than, less than or equal to, greater than, greater than or equal to respectively.

Range Test

This test is used to test whether the value lies within the range or not. The key word is BETWEEN, NOT BETWEEN

Set Membership Test

This is to test whether the value is present in the list of values. The keyword is IN.

Pattern Matching Test

This test is employed to find the name of person starting like Raj. The keyword is LIKE.

%, _ are the wildcard characters used in the pattern matching test. % is equivalent to * in DOS. _(Underscore) is equivalent to ? in DOS. Escape characters $ is used to use legal %, _

Example
WHERE PRODUCT LIKE 'A%BC_'

This will look for first character A Second character anything third and fourth should be B and C respectively. Ending with any number of any characters.

WHERE PRODUCT LIKE '$%%BC_'
This will search for the first character %

Null Value Test

This is to test whether the column contains null value. The keyword is IS NULL.

Compound Search Conditions

This is to test for more than one condition. The keyword AND/OR/NOT is used.

The Where Clause

The WHERE clause is a Boolean expression consisting of column names, column values, relational operators, and logical operators. For example, suppose you have columns Department, Salary, and Bonus. You could use the following WHERE clause to match all employees in the Engineering department that have a salary over 100,000 and a bonus less than 5,000:

```
WHERE Department = 'Engineering ' AND Salary >'100000' AND Bonus <'5000'
SELECT * FROM CUSTOMERS
WHERE CREDIT_LIMIT BETWEEN 10000 AND 50000
SELECT NAME, REP_OFFICE, QUOTA, SALES
WHERE SALES>QUOTA
[ORDER BY columnList2]
```

In the preceding syntax description, columnList1 and columnList2 are comma-separated lists of column names from the table's table1 through table. The SELECT statement returns a result set consisting of the specified columns of the table1 through table, such that the rows of these tables meet the condition of the WHERE clause. If the WHERE clause is omitted, all rows are returned

The ORDER BY clause is used to order the result set by the columns of columnSet2. Each of the column names in the column list may follow by the ASC or DESC keywords. If DESC is specified, the result set is ordered in descending order. Otherwise, the result set is ordered in ascending order.

```
SELECT * FROM CUSTOMERS
ORDER BY COMPANY
```

Union

It is used to combine two or more tables. The necessary criteria is

1. The tables must contain same number of columns.
2. Data type of each should match that of the other
3. Neither can be stored but combined can be stored.

UNION ALL $\longrightarrow$ Produces duplicates

UNION $\longrightarrow$ Default produces Distinct

Example

SELECT * FROM A

UNION (SELECT * FROM B

UNION SELECT * FROM C)

ORDER BY NAME

Simple Joins

Simple joins are required when data is to be retrieved from more than one table.

Example

Order table

```
ORDERNUM ORDER_DATE  CUST  REP  QTY AMOUT
```

Customer Table

```
CUST_NUM COMPANY CUST_REP CREDIT LIMIT
```

List all orders showing 1.Order number 2amount 3 customer name 4customers credit limit

```
SELECT ORDERNUM, AMOUNT, CUST_NUM, CREDIT_LIMIT
FROM ORDER, CUSTOMER
WHERE CUST = CUST_NUM
```

Parent/Children Relationships

In the parent child query, every parent column can have one or more child and every child should have only one parent

In the above example order has one customer and every customer has one or more orders. Therefore Order - Child Customer - Parent

Example

List each salesperson and the city and region where they work

```
SELECT NAME, CITY, REGION
FROM SALESREPS, OFFICES
WHERE REP_OFFICE = OFFICE
```

Foreign Keys

A column in one table whose value matches the primary key in some other table is called a foreign key.

Primary Keys

In a well-designed relational database every database has some column, ore combination of columns whose values uniquely identify each row in the table. This column (or columns) is called primary key of the table.

Once Again Parent Child Relation

The table containing the foreign key is the child in the relationship, the table with the primary key is the parent.

Example

List all offices with target over 5,00,000

```
SELECT CITY, NAME, CITY
FROM SALESREPS, OFFICES
WHERE OFFICE = REP_OFFICE
AND TARGER >=5,00,000
```

Example Multiple matching column

List all the orders showing amounts and product description

```
SELECT ORDER_NUM, PRODUCT, DESCRIPTION, AMOUNT
FROM ORDERS, PRODUCTS
WHERE MFR = MFR_ID   AND
     PRODUCT = PRODUCT_ID
```

Example Quries with three or more tables

List orders over 25,000 including, the name of the salesperson who took the order and the name of the customer who placed it.

```
SELECT NAME, COMPANY, ORDER_NUM, AMOUNT
FROM ORDERS, CUSTOMERS, SALESREPS
WHERE REP = EMPL_NUM AND
CUST = CUST_NUM AND
     AMOUNT > 25000
```

Example Equi Joins

Find all orders received on days when a new salesperson was hired.
```
SELECT ORDER_NUM, ORDER_DATE, AMOUNT, NAME
```

```
FROM ORDERS, SALESREPS
```
WHERE ORDER_DATE = HIRE_DATE

Example Non-Equi Joins

List all combinations of salespeople and offices where the salesperson's quota is more than the office target

Qualified Column names

When two or more table contains the same column name, duplicate column name occurs. An error message will be displayed, when using such columns. To overcome use Tablename.Columnname

When using wildcards use tablename.*

Example Duplicate table using one alias

List of salespeople and their managers

SELECT SALESREPS.NAME, MGR.NAME

FROM SALESREPS, SALESPREPS MGR

WHERE MGR.MANAGER = SALESREPS.EMPL_NUM

The same query using alias table

SELECT REP.NAME, MGR.NAME

FROM SLALESREPS REP, SALESREPS MGR

WHERE MGR.MANAGER = REP.EMPL_NUM

Example

List salespeople with a higher quota than their managers.

```
SELECT SALESREPS.NAME, SALESREPS.QUOTA,
          MGR.QUOTA, MGR.NAME
FROM SALESREPS, SALESREPS.MGR
WHERE SALESREPS.MANAGER = MGR.EMPL_NUM AND
                    SALESREPS.QUOTA > MGR.QUOTA
```

Rules for Multi Table Query Processing

1. If the statement is a UNION of SELECT statements, apply step 2 to 5 to each statement to generate their individual query results.

2. From the product of the tables named in the FROM clause names a single table, the product is that table

3. If there is a WHERE clause, apply its search condition to each row of this product table, retaining those rows for which the search condition is true and discard for FALSE or NULL.

4. For each remaining row, calculate the value of each item in the select list to produce a single row of query results. For each column references, use the value of the column in the current row.

5. If SELECT DISTINCT is specified, eliminate any duplicate rows of query results that were produced.

6. If the statement is a UNION of select statements merge the query results for individual statements into a single table of query results. Eliminate duplicate rows unless union all is specified.

7. If there is an ORDER BY clause, sort the query results as specified.

2.5 Functions

Functions are used to act upon single column or multicolumn for a specified job. Some of the important functions are

SUM()   totals the column
AVG()   average of the column
MIN()   returns the minimum value of the column
MAX()   returns the maximum value of the column
COUNT()   counts the specified condition
COUNT(*)   counts the number of elements in the column

Example

Calculate the total orders for each customer of each salesperson, sorted by customer of each salesperson, sorted by customer, and within each customer by salesperson.

```
SELECT CUST, REP, SUM(AMOUNT)
FROM ORDERS
ORDER BY CUST, REP
```

Example

Calculate the total orders for each customer of each salesperson, sorted by salesperson, and within each salesperson by customer.

```
SELECT REP, CUST, SUM (AMOUNT)
FROM ORDERS
ORDER BY REP, CUST
COMPUTE SUM (AMOUNT) BY REP,
COMPUTE SUM (AMOUNT), AVG (AMOUNT) BY REP
```

Query processing rules with HAVING

If the statement is a union of SELECT statements, apply steps 2 to 7 to each of the statements to generate their individual query results.

From the product of the tables named in the FROM clause. If the FROM clause names a single table, the product is that table.
If there is a WHERE clause, apply its search condition to each row of product table retaining those rows for which the search condition is TRUE discarding those FALSE or NULL.

If there is a group by clause arrange the remaining rows of the product table into row groups, so that the rows in each group have identical values in all of the grouping columns.

If there is a HAVING clause, apply its search condition to each row group, retaining those groups for which the search condition is TRUE

For; each remaining row or row group calculate the value of each item in the select list to produce a single row of query results. For a simple column reference, use the value of the column in the current row. For a column function, use the current row group as its argument if GROUP BY is specified otherwise use the entire set of rows.

If SELECT DISTINCT is specified, eliminate any duplicate rows of query results that were produced.

If the statement is a union of SELECT statement, merge the query results for the individual statements into a single table of query results. Eliminate duplicate rows unless UNION ALL is specified.

If there is an ORDER BY clause, sort the query results as specified.

2.6 Short Summary

- WHERE clause is Boolean expression consisting of column names, column values, relational operators, and logical operators.

- The CREATE DATABASE statement is not supported by all SQL implementations.

- The set methods of the PreparedStatement interface are used to convert Java types into SQL data types.

2.7 Brain Storm

1. How SQL can be used as a Data Query Language?
2. How SQL can be used for Data Manipulation?
3. Write short notes on Functions?
4. What are the Rules for Multi Table Query Processing?

Lecture 3

JDBC Architecture

Objectives

In this lecture you will learn the following

- ODBC and JDBC Drivers
- Microsoft JDBC
- JDBC
- Drivers and their types

<div style="border:2px solid black; padding:10px;">

Coverage Plan

</div>

Lecture 3

3.1 Snap Shot - Remote Database Access

Most useful databases are accessed remotely. In this way, shared access to the database can be provided to multiple users at the same time. For example, you can have a single database server that is used by all employees in the accounting department.

In order to access databases remotely, users need a database client. A database client communicates to the database server on the user's behalf. It provides the user with the capability to update with new information or to retrieve information from the database. In this book, you'll learn to write Java applications and applets that serve as database clients. Your database clients talk to database servers using SQL statements   (See Figure 3.1.)
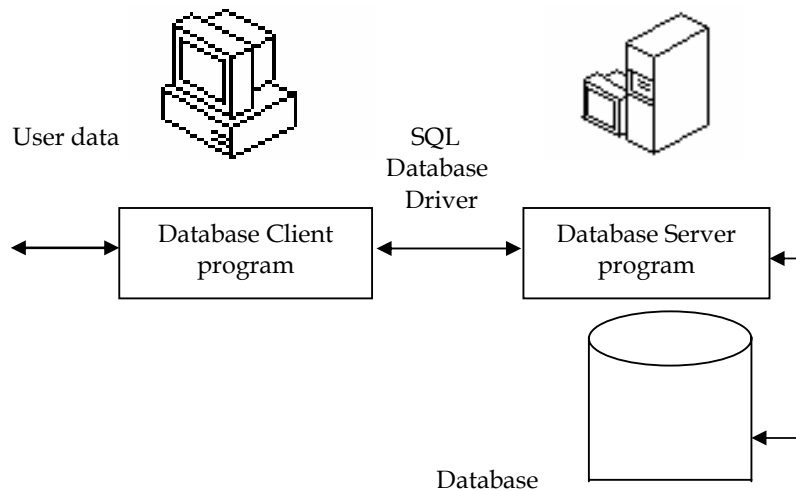
Figure 3.1. A Database client talks to a Database server on the user's behalf.

3.2 ODBC and JDBC drivers

Database clients use database drivers to send SQL statements to database servers and to receive result set and other responses from the servers. JDBC drivers are used by Java applications and applets to communicate with database servers. Officially, Sun says that JDBC is an acronym that does not stand for anything. However, it is associated with "Java database connectivity".

3.3 Microsoft's ODBC

Many database servers use vendor-specific protocols. This means that a database client has to learn a new language to talk to a different database server. However, Microsoft established a common standard for communicating with databases, called Open Database Connectivity (ODBC). Until ODBC, most database clients were server-specific. ODBC drivers abstract away vendor-specific protocols, providing a common application-programming interface to database clients. By writing your database clients to the ODBC API, you enable your programs to access more database servers. (See Figure 3.2)
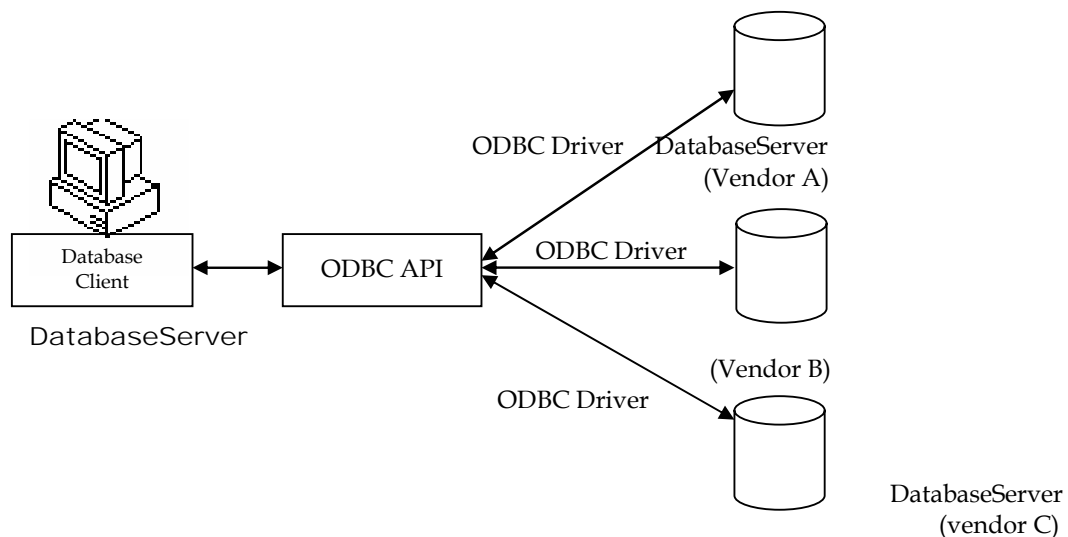
Figure 3.2 A Database client can talk to many database servers via ODBC drivers.

Comes JDBC …

JDBC provides a common database-programming API for Java programs. However, JDBC drivers do not directly communicate with as many databases using ODBC. In fact, one of the first JDBC drivers was the JDBC-ODBC bridge driver developed by JavaSoft and Intersolv. Why did JavaSoft create JDBC? What boil down to the simple fact that ODBC is a better solution for Java applications and applets:

- ODBC is a C language API, not a Java (object-oriented and C is not) API. C uses pointers and other "dangerous" programming constructs that Java does not support. A Java version of ODBC would require a significant rewrite of the ODBC API.
- ODBC drivers must be installed on client machines. This means that applet access to databases would be constrained by the requirement to download and install a JDBC

driver. A pure solution allows JDBC drivers to be automatically downloaded and installed along with applet. This greatly simplifies database access for applet users.

Since the release of the JDBC API, a number of JDBC drivers have been developed. These drivers provide varying levels of capability. JavaSoft has classified JDBC drivers into the following four types:

1. JDBC-ODBC bridge plus ODBC driver- This driver category refers to the original JDBC-ODBC bridge driver. The JDBC-ODBC bridge driver uses Microsoft's ODBC driver to communicate with database servers. It is implemented in both binary code and Java and must be preinstalled on a client computer before it can be used.

JavaSoft created the Java-ODBC bridge driver as a temporary solution to database connectivity until suitable JDBC drivers were developed. The JDBC-ODBC bridge driver translates the JDBC API into the ODBC API and it used with an ODBC driver. The JDBC-ODBC bridge driver is not an elegant solution, but it allows Java developers to use existing ODBC drivers. (See Figure 3.3). The JDBC-ODBC Bridge lets Java clients talk to databases via ODBC drivers.
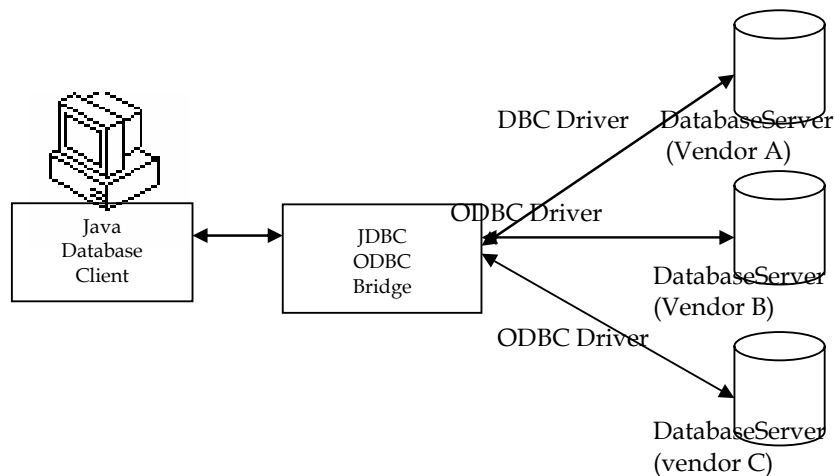
Figure 3.3 A Database client can talk to many database servers via JDBC ODBC drivers.

2. Native-API partly Java driver (also called Type 2 Driver) - This driver category consists of drivers that talk to database servers in the server's native protocol. For example, an Oracle driver would speak SQLNet, while a DB2 driver would use an IBM database protocol. These drivers are implemented in a combination of binary code and Java, and they must be installed on client machines.

A type 2 JDBC driver uses a vendor-specific protocol and must be installed on client machines.

| Java Database Client Database Server | ↔ | Type 2 JDBC Driver | ← | Database Server |

Vendor
Specific Protocol
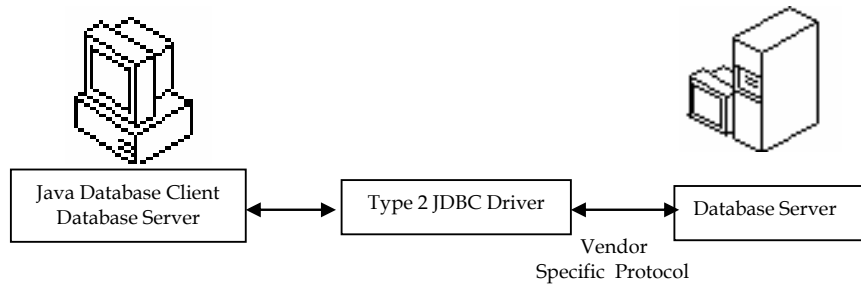
Figure 3.4 A Database client can talk to many database Servers via JDBC type2 drivers.

3. JDBC-Net pure Java driver- This driver category consists of pure Java drivers that speak a standard network protocol (such as HTTP) to a database access server. The database access server then translates the network protocol into a vendor-specific database protocol (possibly using an ODBC driver).
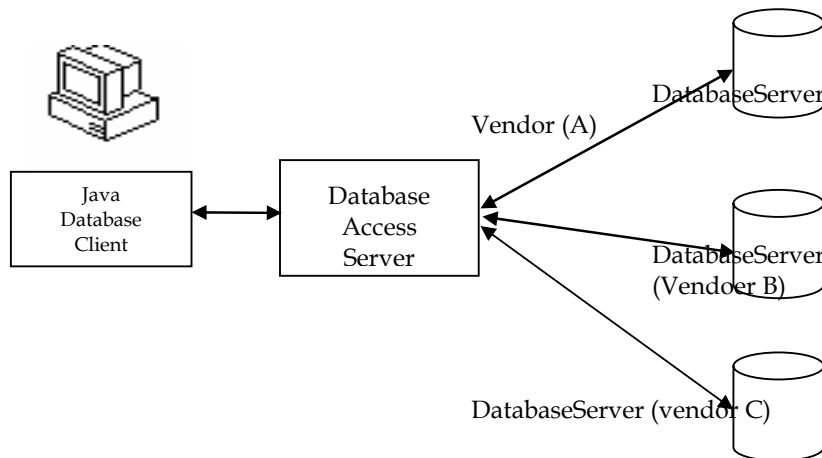
Figure 3.5 A Database client can talk to many database Access servers via JDBC drivers.

Native-protocol pure Java driver- This driver category consists of a pure Java driver that speaks the vendor-specific database protocol of the database server that it is designed to interface with

A type 4 JDBC driver is a pure Java driver that uses a vendor-specific protocol to talk to database servers.
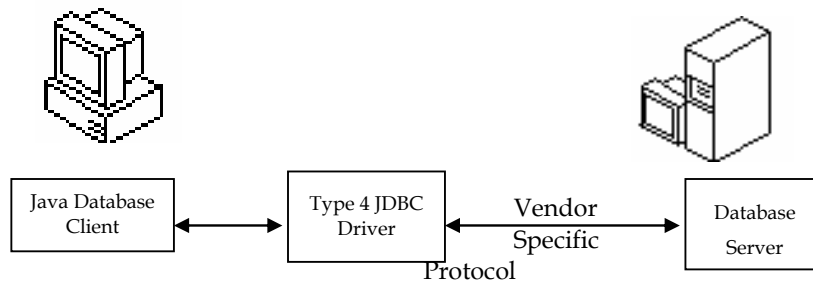
Figure 3.6 A Database client can talk to many database Servers via JDBC type4 drivers.

Of the four types of drivers, only Type 3 and Type 4 are pure Java drives. This is important to suport zero installation for applets. The Type 4 driver communicates with the database server using a vendor-specific protocol, such as SQLNet. The Type 3 driver make as use of a separate database access server. It communicates with the database access server using a standard network protocol,such as HTTP. The database access server communicates with database servers using vendor-specific protocols or ODBC drivers. The IDS JDBC driver Connect to Databases with the java.sql.Package.

3.4 Short Summary

- JDBC provides a common database-programming API for Java programs.

- Database clients use database drivers to send SQL statements to database servers and to receive result set and other responses from the servers.

- JDBC drivers are used by Java applications and applets to communicate with database servers.

3.5 Brain Storm

1. Write short notes on JDBC and ODBC drivers

�...രു

Lecture 4

JDBC

Objectives

In this lecture you will learn the following

- ✎ Connecting to an ODBC Data Source

- ✎ JDBC Implementation

- ✎ Using the Statements

- ✎ ResultSet Processing

Coverage Plan

Lecture 4

4.1 Snap Shot

JDBC is a Java database connectivity API that is a part of the Java Enterprise APIs from Java Soft. From a developer's point of view, JDBC is the first standardized effort to integrate relational databases with Java programs. JDBC has opened all the relational power that can be mustered to Java applets and applications.

Model

JDBC is designed on the CLI model. JDBC defines a set of API objects and methods to interact with the underlying database. A Java program first opens a connection to a database, makes a statement object, passes SQL statements to the underlying DBMS through the statement object, and retrieves the results as well as information about the result sets. Typically, the JDBC class files and the Java applet reside in the client. To minimize the latency during execution, it is better to have the JDBC classes in the client.

As a part of JDBC, Java Soft also delivers a driver to access ODBC data sources from JDBC. This driver is jointly developed with Intersolv and is called the JDBC-ODBC bridge. The JDBC-ODBC bridge is implemented as the JdbcOdbc.class and a native library to access the ODBC driver. For the Windows platform, the native library is a DLL (JDBCODBC.DLL).

As JDBC is close to ODBC in design, the ODBC Bridge is a thin layer over JDBC. Internally, this driver maps JDBC methods to ODBC calls and, thus, interacts with any available ODBC driver. The advantage of this bridge is that now JDBC has the capability to access almost all databases, as ODBC drivers are widely available. The JDBC-ODBC Bridge allows JDBC driver to be used as ODBC drivers by converting JDBC method calls into ODBC function calls.

Drivers

Using the JDBC-ODBC Bridge requires three things:

The JDBC-ODBC bridge driver included with Java2:

`sun.jdbc.odbc.JdbcOdbcDriver` an ODBC driver.

An ODBC data source that has been associated with the driver using software such as the ODBC Data Source Administrator.

ODBC data sources can be set up from within some database programs. When a new database file is created in any ODBC supported application system, users have the option of associating it with an ODBC driver.

All ODBC data sources must be given a short descriptive name. This name will be used inside Java programs when a connection is made to the database that the source refers to. In Windows environment, once an ODBC driver is selected and the database is created, they will show up in the ODBC Data Source Administrator.

4.2 Connecting to an ODBC Data Source

Java application that uses a JDBC-ODBC bridge to connect to a database file either a dbase, Excel, FoxPro, Access, SQL Server, Oracle and many more.

First open the ODBC Data Source 32Bit from the Control Panel. Switch to System DSN. System DSN lists the System Data Sources. This list shows all system DSNs, including the name of each DSN and the driver associated with the DSN. Click the Add button to create new Data Source See figure (4.1)
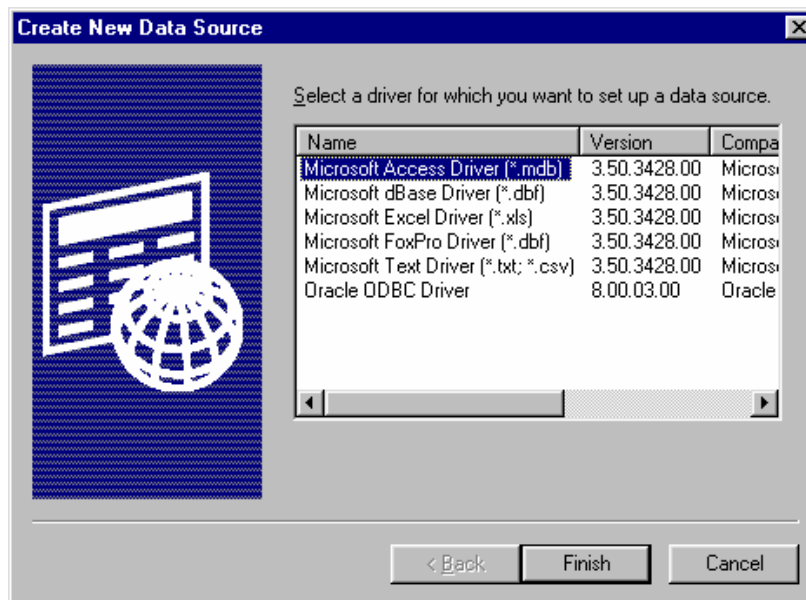
Figure 4.1 Create New Data Source

Select Oracle ODBC Driver or any other and click the Finish button to finish. This will pop up with a new window Oracle8 ODBC Driver Setup. Give the Data Source name as oraodbc, UserID Scott and click OK to finish. (See figure 4.2)
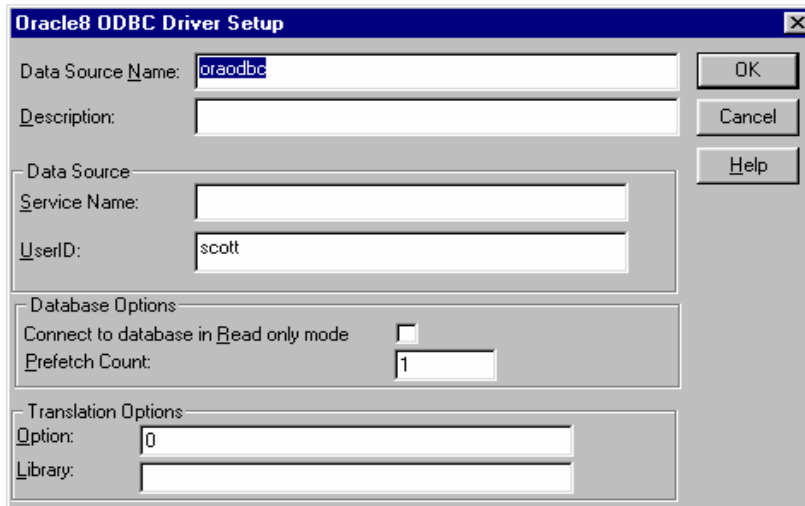
Figure 4.2 Oracle8 ODBC Driver Setup
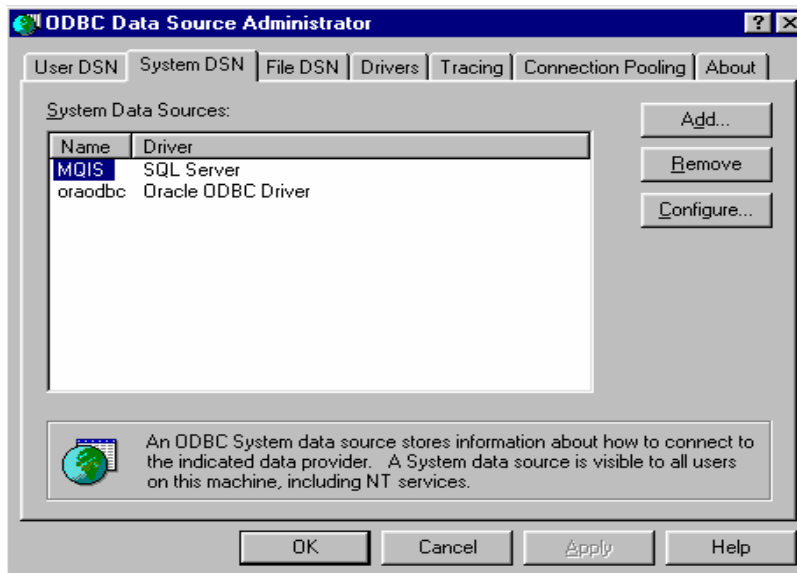
Figure 4.3 ODBC Data Source Administrator

After finishing Oracle8 ODBC Driver set up, the ODBC Data Source Administrator will be displaying the following. (See figure 4.3). Now the ODBC Driver Connection has been established. To display the driver-specific data source setup dialog box for a user data source, double-click the system DSN.

Now the dialog box will be look as figure 4.3. Click OK buttons to Complete the Connection.

4.3 JDBC Connection

Sun offers a package java.sql that allows java program to access relational database management systems (RDBMS). Through the JDBC a relational database can be accessed using the sql. To communicate with a relational database the following steps have to be followed.

Establish a connection between the Java program and the database manager. Send a sql statement to the database by using a statement object. Read the results back from the database and use them in the program.

Working with the driver manager

JDBC is designed to work with many different database managers from different applications. In order to establish a connection with a database the Java runtime environment must load the driver for the specified database. The driver manager class is responsible for loading and unloading drivers.

Loading drivers

JDBC drivers are typically written by a database vendor; they accept JDBC connections and statements from one side and issue native calls to the database from the other. To use a JDBC driver (including the JDBC-ODBC bridge driver) first thing is to loading it.

Preloading from Command line

The command for the command line is

Java –Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver orajdbc

Preloading from program

The command line command will not be of much use for programs and applications, a sample programmatic version is given below. Usually the following statements will form the first block in the JDBC programming.

```
try
{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}

catch(ClassNotFoundException e)
{
System.err.println("Unable to find JDBC driver");
}
```

4.4 JDBC Implementation

JDBC is implemented as the java.sql package. This package contains all of the JDBC classes and methods, as shown in Table 4.1.

Type	Class
Driver	Java.sql.Driver java.sql.DriverManager java.sql.DriverPropertyInfo
Connection	Java.sql.Connection
Statements	Java.sql.Statement java.sql.PreparedStatement java.sql.CallableStatement
ResultSet	java.sql.ResultSet
Errors/Warning	java.sql.SQLException java.sql.SQLWarning
Metadata	java.sql.DatabaseMetaData java.sql.ResultSetMetaData
Date/Time	java.sql.Date java.sql.Time java.sql.Timestamp
Miscellaneous	Java.sql.Types java.sql.DataTruncation

Table 4.1 JDBC Classes

Using the connection class

Once a driver has registered with the DriverManager connection to a database is made simple. To invoke the driver and return a reference to a connection a new connection to be

made. First specify the location of the database, then the user name and password. The sample code for connection is

```
Connection myConnection = DriverManager.getConnection
("jdbc:odbc:MyDataSource", "Administrator",
"Password");
```

When DriverManager gets a getconnection() request, it takes the JDBC URL and passes it to each registered Driver in turn. The first Driver to recognize the URL and say that it can connect gets to establish the connection. If no Driver can handle the URL, DriverManager throws a SQLException, reporting "no suitable driver". To check whether Driver has been installed correctly, check for throwsexception error.

Use the connection object to connect to databases. By default, the new connection is set to auto-commit every statement is instantly committed to the database.

DBC URL

True to the nature of the Internet, JDBC identifies a database with an URL. The URL is of the form:

jdbc:<subprotocol>:<subname related to the DBMS/Protocol>
For databases on the Internet or intranet, the subname can contain the Net URL *//hostname:port/* The <subprotocol> can be any name that a database understands. The odbc subprotocol name is reserved for ODBC style data sources. A normal ODBC database JDBC URL looks like the following:

jdbc:odbc:<ODBC DSN>;User=<username>;PW=<password>

To develop a JDBC driver with a new subprotocol, it is better to reserve the subprotocol name with JavaSoft, which maintains an informal subprotocol registry.

Return Type	Method Name	Parameter
java.sql.Driver		
Connection	connect	(String url, java.util.Properties info)
Boolean	AcceptsURL	(String url)

DriverPropertyInfo[]	GetPropertyInfo	(String url, java.util.Properties info)
Int	GetMajorVersion	()
Int	getMinorVersion	()
Boolean	jdbcCompliant	()
java.sql.DriverManager		
Connection	getConnection	(String url, java.util.Properties info)
Connection	getConnection	(String url, String user, String password)
Connection	getConnection	(String url)
Driver	getDriver	(String url)
void	registerDriver	(java.sql.Driver driver)
void	deregisterDriver	(Driver driver)
java.util.Enumeration	getDrivers	()
void	setLoginTimeout	(int seconds)
int	getLoginTimeout	()
void	setLogStream	(java.io.PrintStream out)
java.io.PrintStream	getLogStream	()
Void	println	(String message)
Class Initialization Routine		
Void	Initialize	()

Table 4.2 Driver, DriverManager and Related Methods

The Connection class is one of the major classes in JDBC. It packs a lot of functionality, ranging from transaction processing to creating statements, in one class as seen in Table 4.3.

Return Type	**Method Name**	**Parameter**
Statement-Related Methods		
Statement	createStatement	()
PreparedStatement	prepareStatement	(String sql)
CallableStatement	prepareCall	(String sql)
String	nativeSQL	(String sql)
void	close	()

Boolean	isClosed	()
Metadata-Related Methods		
DatabaseMetaData	getMetaData	()
void	SetReadOnly	(boolean readOnly)
Boolean	IsReadOnly	()
void	setCatalog	(String catalog)
String	getCatalog	()
SQLWarning	getWarnings	()
void	clearWarnings	()
Transaction-Related Methods		
void	setAutoCommit	(booleanautoCommit)
Boolean	getAutoCommit	()
void	commit	()
void	rollback	()
void	SetTransactionIsolation	(int level)
Int	GetTransactionIsolation	()

Table 4.3 java.sql.Connection Methods and Constants

Managing SQL transactions

Use connection's set autocommit() method to disable auto_commit.

```
Issue sql statements
```

To commit the changes to the database, call commit () the transactions. To abandon all the statements made since last commit (), call rollback ().

Note: By default the Connection automatically commits changes after executing each statement. If auto commit has been disabled, an explicit commit must be done or database changes will not be saved.

Using the Statement

Use a Statement object to hold sql statements. When a statement object is send to the database, the database runs the sql and returns a ResultSet.

```
Resultset   myset   =   mystatement.executeQuery("SELECT   *   FROM
CUSTOMERS");
Or by
Resultset myset;
if(mystatement.execute("SELECT * FROM CUSTOMERS"))
myset = mystatement.getResultset();
```

The execute () method returns a Boolean true if the Statement returned a ResultSet and false if it returned an integer. The execute () method is included in JDBC 2.0 and we will be using executequery () and executeupdate () and will be discussed later.

Statement

A Statement object is created using the createStatement() method in the Connection object. Table 4.4 shows all methods available for the Statement object.

Return Type	Method Name	Parameter
ResultSet	executeQuery	(String sql)
int	executeUpdate	(String sql)
boolean	execute	(String sql)
boolean	getMoreResults	()
void	close	()
int	getMaxFieldSize	()
void	setMaxFieldSize	(int max)
int	getMaxRows	()
void	setMaxRows	(int max)
void	setEscapeProcessing	(boolean enable)
int	getQueryTimeout	()
void	setQueryTimeout	(int seconds)
void	cancel	()

java.sql.SQLWarning	getWarnings	()
void	clearWarnings	()
void	setCursorName	(String name)
ResultSet	getResultSet	()
Int	getUpdateCount	()

Table 4.4 Statement Object Methods

The most important methods are executeQuery (), executeUpdate () and execute(). When creating a Statement object with a SQL statement, the executeQuery () method takes a SQL string. It passes the SQL string to the underlying data source through the driver manager and gets the ResultSet back to the application program. The executeQuery () method returns only one ResultSet. For those cases that return more than one ResultSet, the execute () method should be used.

Complete Program

Now we shall look at some complete programs. sql stament to to create a table named customers.

```
CREATE TABLE CUSTOMERS
(
CUST_NUM          INTEGER,
COMPANY           VARCHAR(20),
CUST_REP          INTEGER,
CREDIT_LIMIT NUMBER(7,2)
)
```

The following Java Program can create the same table Customers

```
import java.sql.*;
import java.io.*;

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

```
        {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");


      Statement stmt = con.createStatement();
stmt.executeUpdate("create table
customers(CUST_NUM int, COMPANY
char(20), CUST_REP int, CREDIT_LIMIT number(7,2))");


            stmt.close();
            con.close();
      System.out.println("Table Successfully created");
      }catch(Exception e)
      {
            e.printStackTrace();
      }
    }
}
```

The output of the program is

Table Successfully created

First statement import java.sql.* ; imports all classes that belong to the package sql. All the JDBC code will be delimited in the try block to avoid any exceptional handling.

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Specifies the type of driver to the JRT as JdbcOdbcDriver

Class.forName("my.sql.Driver");

When the method getConnection is called, the DriverManager will attempt to locate a suitable driver from amongst those loaded at initialization and those loaded explicitly using the same classloader as the current applet or application.

Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");

Creates Connection object named con.

A Connection's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on. This information is obtained with the getMetaData method.

DriverManager

As part of its initialization, the DriverManager class will attempt to load the driver classes referenced in the "jdbc.drivers" system property. This allows a user to customize the JDBC Drivers used by their applications.

Getconnection

Attempts to establish a connection to the given database URL. The DriverManager attempts to select an appropriate driver from the set of registered JDBC drivers.

Parameters:

url - a database url of the form jdbc:subprotocol:subname
info - a list of arbitrary string tag/value pairs as connection arguments; normally at least a "user" and "password" property should be included
Returns:

a Connection to the URL

Statement stmt = con.createStatement();

The object used for executing a static SQL statement and obtaining the results produced by it.

Only one ResultSet per Statement can be open at any point in time. Therefore, if the reading of one ResultSet is interleaved with the reading of another, each must have been generated by different Statements. All statement execute methods implicitly close a statment's current ResultSet if an open one exists.

CreateStatement

Creates a Statement object for sending SQL statements to the database. SQL statements without parameters are normally executed using Statement objects. If the same SQL statement is executed many times, it is more efficient to use a PreparedStatement JDBC 2.0 Result sets created using the returned Statement will have forward-only type, and read-only concurrency, by default.

Returns: a new Statement object

ExecuteUpdate

Executes an SQL INSERT, UPDATE or DELETE statement. In addition, SQL statements that return nothing, such as SQL DDL statements, can be executed.

Parameters:

sql - a SQL INSERT, UPDATE or DELETE statement or a SQL statement that returns nothing

Returns:

either the row count for INSERT, UPDATE or DELETE or 0 for SQL statements that return nothing

stmt.close();

Releases this Statement object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

con.close();

Releases a Connection's database and JDBC resources immediately instead of waiting for them to be automatically released.

Every object in sql package throws an exception. It is handled by the try catch(e Exception) block.

An exception that provides information on a database access error.

Each SQLException provides several kinds of information:

a string describing the error. This is used as the Java Exception message, available via the method getMessage ().

a "SQLstate" string, which follows the XOPEN SQLstate conventions. The values of the SQLState string are described in the XOPEN SQL spec.

an integer error code that is specific to each vendor. Normally this will be the actual error code returned by the underlying database.

a chain to a next Exception. This can be used to provide additional error information.

e.printStackTrace();

Prints a message to the current JDBC log stream.

Selecting Rows

Selecting Rows from Customers table has the following sql code.

```
SELECT * FROM CUSTOMERS
```

Gives the following result. And the equivalent java code is given below.

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
101	NY TRADERS	5009	40000
102	RAVI AND CO	5008	50000
103	RAM BROTHERS	5007	23000

```java
import java.sql.*;
import java.io.*;
class SelectRow
{
    public static void main(String[] args)
    {
```

```
        ResultSet rs;


        try
        {
              Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");
              Statement stmt = con.createStatement();
        rs = stmt.executeQuery("select * from
CUSTOMERS");


        System.out.println("CUST_NUM" + "\tCOMPANY" +
"\t\tCUST_REP" + "\tCREDIT_LIMIT");


              while(rs.next())
              {
    System.out.println(rs.getInt("CUST_NUM") + "\t\t" +
rs.getString("COMPANY")+    "\t"+   rs.getInt("CUST_REP")   +   "\t"   +
rs.getInt("CREDIT_LIMIT"));


              }
              stmt.close();
              con.close();
System.out.println("Records successfully selected");
        }catch(Exception e)
        {
              e.printStackTrace();
        }
    }
}
```

The output of the program is

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
101	NY TRADERS	5009	40000
102	RAVI AND CO	5008	50000
103	RAM BROTHERS	5007	23000

Records successfully selected

4.5 Resultset Processing: Retrieving Results

Here we are using a new object ResultSet. A ResultSet provides access to a table of data. A ResultSet object is usually generated by executing a Statement. The ResultSet object is actually a tabular data set; that is, it consists of rows of data organized in uniform columns. Table 4.5 shows the methods associated with the ResultSet object.

In JDBC, the Java program can see only one row of data at one time. The program uses the next() method to go to the next row. JDBC does not provide any methods to move backwards along the ResultSet or to remember the row positions (called *bookmarks* in ODBC). A ResultSet maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The 'next' method moves the cursor to the next row. So in this program we are using while(rs.next()). The method rs.next() returns a boolean value depending on the recordset. If it reaches last record false in returned and loop lapses.

For maximum portability, ResultSet columns within each row should be read in left-to-right order and each column should be read only once.

The JDBC driver attempts to convert the underlying data to the specified Java type and returns a suitable Java value through the getXXX methods. See the JDBC specification for allowable mappings from SQL types to Java types with the ResultSet.getXXX methods.

Column names used as input to getXXX methods are case insensitive. When performing a getXXX using a column name, if several columns have the same name, then the value of the first matching column will be returned. The column name option is designed to be used when column names are used in the SQL query. For columns that are NOT explicitly named in the query, it is best to use column numbers. If column names are used, there is no way for the programmer to guarantee that they actually refer to the intended columns.

A ResultSet is automatically closed by the Statement that generated it when that Statement is closed, re-executed, or used to retrieve the next result from a sequence of multiple results.

Return Type	Method Name	Parameter
Boolean	next	()
void	close	()
Boolean	wasNull	()
Get Data By Column		

Position		
java.io.InputStream	getAsciiStream	(int columnIndex)
java.io.InputStream	getBinaryStream	(int columnIndex)
boolean	getBoolean	(int columnIndex)
byte	getByte	(int columnIndex)
byte[]	getBytes	(int columnIndex)
java.sql.Date	getDate	(int columnIndex)
double	getDouble	(int columnIndex)
float	getFloat	(int columnIndex)
int	getInt	(int columnIndex)
long	getLong	(int columnIndex)
java.lang.Bignum	getBignum	(int columnIndex, int scale)
Object	getObject	(int columnIndex)
short	getShort	(int columnIndex)
String	getString	(int columnIndex)
java.sql.Time	getTime	(int columnIndex)
java.sql.Timestamp	getTimestamp	(int columnIndex)
java.io.InputStream	getUnicodeStream	(int columnIndex)
Get Data By Column Name		
Java.io.InputStream	getAsciiStream	(String columnName)
Java.io.InputStream	getBinaryStream	(String columnName)
Boolean	getBoolean	(String columnName)
Byte	getByte	(String columnName)
Byte[]	getBytes	(String columnName)
java.sql.Date	getDate	(String columnName)
double	getDouble	(String columnName)
float	getFloat	(String columnName)
int	getInt	(String columnName)
long	getLong	(String columnName)
java.lang.Bignum	getBignum	(String columnName, int scale)
Object	getObject	(String columnName)

short	getShort	(String columnName)
String	getString	(String columnName)
java.sql.Time	getTime	(String columnName)
java.sql.Timestamp	getTimestamp	(String columnName)
java.io.InputStream	getUnicodeStream	(String columnName)
int	findColumn	(String columnName)
SQLWarning	getWarnings	()
void	clearWarnings	()
String	getCursorName	()
ResultSetMetaData	getMetaData	()

Table 4.5 java.sql.ResultSet Methods

The ResultSet methods even though there are many are very simple. The major ones are the getXXX() methods. The getMetaData() method returns the meta data information about a ResultSet. The DatabaseMetaData also returns the results in the ResultSet form. The ResultSet also has methods for the silent SQLWarnings. It is a good practice to check any warnings using the getWarning() method that returns a null if there are no warnings.

Once the program has a row, it can use the positional index (1 for the first column, 2 for the second column, and so on) or the column name to get the field value by using the getXXXX() methods.

The getXXX methods retrieve column values for the current row. Retrieve values using either the index number of the column or the name of the column. In general, using the column index will be more efficient. Columns are numbered from 1.

Here is the revised version of Selecting Row using the column number

```
import java.sql.*;
import java.io.*;

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

```
        ResultSet rs;
        try
        {
             Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
             Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");
             Statement stmt = con.createStatement();
rs = stmt.executeQuery("select * from CUSTOMERS");

     System.out.println("CUST_NUM" + "\tCOMPANY" +
"\t\tCUST_REP" + "\tCREDIT_LIMIT");
     while(rs.next())
      {
        int no=rs.getInt(1);
        String company=rs.getString(2);
        int rep=rs.getInt(3);
        double credit=rs.getDouble(4);
         System.out.println(no+"\t\t"+company+"\t"+rep+"\t\t"+credit);

      }
             stmt.close();
             con.close();
     System.out.println("Records successfully selected");
     }catch(Exception e)
     {
             e.printStackTrace();
     }
    }
}
```

Deleting a Record

In thin proogram records are being deleted from the customers. Here we are using both the executeUpdate() and executeQuery() to deleting and retriving records from the Customers table. The executeUpdate() returns void and executeQuery() returns a ResultSet object. Here is the complete program.

//PROGRAM TO DELETE A RECORD

```
import java.sql.*;
import java.io.*;
```

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

          ResultSet rs;
          try
          {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");
       Statement stmt = con.createStatement();
stmt.executeUpdate
    ("delete from CUSTOMERS where  CUST_NUM = 101");
rs = stmt.executeQuery("select * from customers");
       System.out.println("CUST_NUM" + "\tCOMPANY" +
"\t\tCUST_REP" + "\tCREDIT_LIMIT");
              while(rs.next())
              {
                     int no=rs.getInt(1);
                     String company=rs.getString(2);
                     int rep=rs.getInt(3);
                     double credit=rs.getDouble(4);
System.out.println( no + "\t\t"+company + "\t" +     rep + "\t\t" + credit);

              }
              stmt.close();
              con.close();
System.out.println("Record Successfully deleted");
       }catch(Exception e)
       {
              e.printStackTrace();
       }
    }
}
```

The result of the program

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
102	RAVI AND CO	5008	50000.0
103	RAM BROTHERS	5007	23000.0

Record successfully deleted

Inserting values in to the database. This program is very similar to the delete records program.

Inserting a Record

//PROGRAM FOR INSERTING A RECORD

```
import java.sql.*;
import java.io.*;

class Insert
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");
            Statement stmt = con.createStatement();
        stmt.executeUpdate("insert into CUSTOMERS values
(1,'CHARLIE AND CO',20,45000)");
stmt.executeUpdate("insert into CUSTOMERS values
(2,'ARVIND MILLS',30,50000)");
stmt.executeUpdate("insert into CUSTOMERS values
(3,'PANDY BROTHERS',20,12500)");


ResultSet rs = stmt.executeQuery("select * from CUSTOMERS");

System.out.println("CUST_NUM" + "\tCOMPANY" +
"\t\tCUST_REP" + "\tCREDIT_LIMIT");

            while(rs.next())
            {
                    int no=rs.getInt(1);
                    String company=rs.getString(2);
                    int rep=rs.getInt(3);
                    double credit=rs.getDouble(4);
```

```
System.out.println( no + "\t\t"+company + "\t" +    rep    +       "\t\t"    +
credit);


            }
            stmt.close();
            con.close();
System.out.println("Records successfully
inserted");
      }catch(Exception e)
      {
            e.printStackTrace();
      }
    }
}
```

The output of the program

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
2	ARVIND MILLS	30	50000
102	RAVI AND CO	5008	50000
103	RAM BROTHERS	5007	23000
1	CHARLIE AND CO	20	45000
3	PANDY BROTHERS	20	12500

Records successfully inserted

Updating Records

This program updates records in the customer table. Here the credit_limit of the Arvind Mills(whose CUST_NUM is 2) has been updated from 50,000 to 2,00,000

//PROGRAM FOR UPDATING A RECORD

```
import java.sql.*;
import java.io.*;

class UpdateCust
```

```
{
     public static void main(String[] args)
     {
         try
         {
             Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");
             Statement stmt = con.createStatement();
      stmt.executeUpdate("update CUSTOMERS set
CREDIT_LIMIT = 200000 where CUST_NUM =2");

      ResultSet rs = stmt.executeQuery("select * from
CUSTOMERS");

      System.out.println("CUST_NUM" + "\tCOMPANY" +
"\t\tCUST_REP" + "\tCREDIT_LIMIT");

             while(rs.next())
             {
                 int no=rs.getInt(1);
                 String company=rs.getString(2);
                 int rep=rs.getInt(3);
                 double credit=rs.getDouble(4);

      System.out.println(no + "\t\t"+company + "\t" +      rep +   "\t\t"  +
credit);

             }

             stmt.close();
             con.close();
System.out.println("Records successfully updated");
      }catch(Exception e)
      {
             e.printStackTrace();
      }
   }
}
```

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
2	ARVIND MILLS	30	200000

Centre for Information Technology and Engineering, Manonaniam Sundaranar University

102	RAVI AND CO	5008	50000
103	RAM BROTHERS	5007	23000
1	CHARLIE AND CO	20	45000
3	PANDY BROTHERS	20	12500

Records successfully updated

Deleting a Table

//PROGRAM TO DROP A TABLE

```
import java.sql.*;
import java.io.*;

class Drop
{
    public static void main(String[] args)
    {
        try
        {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");
            Statement stmt = con.createStatement();
            stmt.executeUpdate("drop table CUSTOMERS");

ResultSet rs = stmt.executeQuery("select * from
tab");
      System.out.println("TNAME" + "\t\tTABTYPE" +
"\tCLUSTERID");

            while(rs.next())
            {
                    String name = rs.getString(1);
                    String type = rs.getString(2);
                    String clus = rs.getString(3);
System.out.println( name + "\t\t" + type + "\t" + clus);

            }
            stmt.close();
            con.close();
System.out.println("Customer Table successfully dropped");
        }catch(Exception e)
```

```
        {
                e.printStackTrace();
        }
    }
}
```

The output of the program would be similar to this.

TNAME	TABTYPE	CLUSTERID
BONUS	TABLE	null
DEPT	TABLE	null
EMP	TABLE	null
EMP1	TABLE	null
SALGRADE	TABLE	null
SAL_DET	TABLE	null

Customer Table successfully dropped

4.6 Short Summary

- java.sql package of sun allows java program to access relational database management systems.

- JDBC is designed to work with many different database managers from different applications.

- Connection class is the one of the major class in JDBC.

4.7 Brain Storm

1. How to get JDBC connection?
2. How can you implement JDBC?
3. How do you retrieve result?

ஜ…ஐ

Lecture 5

JDBC

Objectives

In this lecture you will learn the following

- ✍ Prepared Statements

- ✍ Callable Statements

- ✍ JDBC Classes

<div style="text-align: center; border: 3px double black;">

Coverage Plan

</div>

Lecture 5

5.1 Snap Shot

This lecture will discuss on Prepared statements, Callable statements, jdbc classes, moving cursors in Scrollable Resultset , making updates to updateable resultsets, updatind a resultset programatically and etc…

5.2 Prepared Statement

In the case of a PreparedStatement object, as the name implies, the application program prepares a SQL statement using the java.sql.Connection.prepareStatement()method.The PreparedStatement() method takes a SQL string, which is passed to the underlying DBMS. The DBMS goes through the syntax run, query plan optimization, and the execution plan generation stages, but does not execute the SQL statement. Possibly, it returns a handle to the optimized execution plan that the JDBC driver stores internally in the PreparedStatement object.

The methods of the PreparedStatement object are shown in Table 4.6 Notice that the executeQuery(), executeUpdate(), and execute() methods do not take any parameters. They are just calls to the underlying DBMS to perform the already-optimized SQL statement.

Return Type	Method Name	Parameter
ResultSet	executeQuery	()
Int	executeUpdate	()
Boolean	execute	()

Table 4.6 Prepared Statement Object Methods

One of the major features of a PreparedStatement is that it can handle IN types of parameters. The parameters are indicated in a SQL statement by placing the "?" as the parameter marker instead of the actual values. In the Java program, the association is made to the parameters with the setXXXX() methods, as shown in Table 4.7 All of the setXXXX() methods take the parameter index, which is 1 for the first "?," 2 for the second "?," and so on.

Return Type	Method Name	Parameter
void	ClearParameters	()
void	SetAsciiStream	(int parameterIndex, java.io.InputStream x, int length)
void	SetBinaryStream	(int parameterIndex, java.io.InputStream x, int length)

void	SetBoolean	(int parameterIndex, boolean x)
void	setByte	(int parameterIndex, byte x)
void	1setBytes	(int parameterIndex, byte x[])
void	setDate	(int parameterIndex, java.sql.Date x)
void	setDouble	(int parameterIndex, double x)
void	setFloat	(int parameterIndex, float x)
void	setInt	(int parameterIndex, int x)
void	setLong	(int parameterIndex, long x)
void	setNull	(int parameterIndex, int sqlType)
void	SetBignum	(int parameterIndex, Bignum x)
void	setShort	(int parameterIndex, short x)
void	setString	(int parameterIndex, String x)
void	setTime	(int parameterIndex, java.sql.Time x)
void	SetTimestamp	(int parameterIndex, java.sql.Timestamp x)
void	SetUnicodeStream	(int parameterIndex, java.io.InputStream x, int length)
Advanced Features— Object Manipulation		
Void	setObject	(int parameterIndex, Object x, int targetSqlType, int scale)
Void	setObject	(int parameterIndex, Object x, int targetSqlType)
Void	setObject	(int parameterIndex, Object x)

Table 4.7 java.sql.PreparedStatement--Parameter-Related Methods

In the case of the PreparedStatement, the driver actually sends only the execution plan ID and the parameters to the DBMS. This results in less network traffic and is well-suited for Java applications on the Internet. The PreparedStatement should be used when needed to execute the SQL statement many times in a Java application. But remember, even though the optimized execution plan is available during the execution of a Java program, the DBMS discards the execution plan at the end of the program. So, the DBMS must go through all of the steps of creating an execution plan every time the program runs. The PreparedStatement object achieves faster SQL execution performance than the simple Statement object, as the DBMS does not have to run through the steps of creating the execution plan.

The following example program shows how to use the PreparedStatement class to access a database. The simple Statement example can be improved in a few major ways. First, the DBMS goes through building the execution plan every time, so make it a PreparedStatement. Secondly, the query lists all courses, which could scroll away.

//PROGRAM FOR PREPARE STATEMENT

```java
import java.sql.*;
import java.io.*;


class PrepareSt
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con = DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");
PreparedStatement ps = con.prepareStatement("Select * from CUSTOMERS
where CREDIT_LIMIT >= ?");
            ps.setInt(1,50000);

            ResultSet rs=ps.executeQuery();
//    ResultSet rs = stmt.executeQuery("select * from
CUSTOMERS");

        System.out.println("CUST_NUM" + "\tCOMPANY" +
"\t\tCUST_REP" + "\tCREDIT_LIMIT");

            while(rs.next())
              {
                    int no=rs.getInt(1);
                    String company=rs.getString(2);
                    int rep=rs.getInt(3);
                    double credit=rs.getDouble(4);

System.out.println(no+"\t\t"+company+"\t"+rep+"\t\t"
+credit);
              }
            rs.close();
            ps.close();
            con.close();

        }catch(Exception e)
        {
```

```
                    e.printStackTrace();
            }
        }
}
```

The output of the program may be equal to

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
2	ARVIND MILLS	30	900000.0

5.3 Callable Statement

For a secure, consistent, and manageable multi-tier client/server system, the data access should allow the use of stored procedures. Stored procedures centralize the business logic in terms of manageability and also in terms of running the query. JDBC allows the use of stored procedures by the CallableStatement class and with the escape clause string.

A CallableStatement object is created by the prepareCall() method in the Connection object. The prepareCall() method takes a string as the parameter. This string, called an *escape clause*, is of the form

{[? =] call <stored procedure name>
[<parameter>,<parameter> ...]}

The CallableStatement class supports parameters. These parameters are of the OUT kind from a stored procedure or the IN kind to pass values into a stored procedure. The parameter marker (question mark) must be used for the return value (if any) and any output arguments because the parameter marker is bound to a program variable in the stored procedure. Input arguments can be either literals or parameters. For a dynamic parameterized statement, the escape clause string takes the form.

{[? =] call <stored procedure name> [<?>,<?> ...]}
The OUT parameters should be registered using the registerOutparameter() method as shown in Table 4.8 before the call to the executeQuery(), executeUpdate(), or execute() methods.

Return Type	Method Name	Parameter
Void	RegisterOutParameter	(int parameterIndex, int sqlType)
Void	RegisterOutParameter	(int parameterIndex, int sqlType, int scale)

Table 4.8 *CallableStatement--OUT* **Parameter Register Methods**

After the stored procedure is executed, the DBMS returns the result value to the JDBC driver. This return value is accessed by the Java program using the methods in Table 4.9

Return Type	Method Name	Parameter
Boolean	getBoolean	(int parameterIndex)
Byte	getByte	(int parameterIndex)
byte[]	getBytes	(int parameterIndex)
java.sql.Date	getDate	(int parameterIndex)
double	getDouble	(int parameterIndex)
float	getFloat	(int parameterIndex)
int	getInt	(int parameterIndex)
long	getLong	(int parameterIndex)
java.lang.Bignum	getBignum	(int parameterIndex, int scale)
Object	getObject	(int parameterIndex)
short	getShort	(int parameterIndex)
String	getString	(int parameterIndex)
java.sql.Time	getTime	(int parameterIndex)
java.sql.Timestamp	getTimestamp	(int parameterIndex)
Miscellaneous Functions		
boolean	wasNull	()

Table 4.9 CallableStatement Parameter Access Methods

Consider a table of emp1 with the following structure.

EMPNO	NAME	AGE	SAL	DEPTNO
101	aaa	24	9400	10
102	bbb	25	12400	20
103	ccc	27	11400	30

Creating a procedure

The code given below crates a procedure named proc, which increases the credit limit by n.

```
create or replace procedure proc(n number)
as
begin
update customers set CREDIT_LIMIT = CREDIT_LIMIT + n;
commit;
end;
```

//Program for procedure

```
import java.sql.*;
import java.io.*;

public class Procedure
{
  public static void main(String args[])
  {
    try
    {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con=DriverManager.getConnection
("jdbc:odbc:oraodbc","scott","tiger");

   System.out.print("Enter Credit limit increment:");
BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
      String str=br.readLine();
```

```
        int p=Integer.parseInt(str);


CallableStatement cs=con.prepareCall("{ call proc(?)
}");
        cs.setInt(1,100);
        cs.execute();
        System.out.println("Procedure Executed");
        cs.close();


        Statement st = con.createStatement();
        ResultSet rs=st.executeQuery("Select * from
        Customers");


         while(rs.next())
          {
            int eno=rs.getInt(1);
            String name=rs.getString(2);
            int ag=rs.getInt(3);
            double sal=rs.getDouble(4);
System.out.println(eno+"\t"+name+"\t"+ag+"\t"+sal);
          }
        rs.close();
        st.close();
        con.close();


     }
     catch(SQLException es)
     {
       System.out.println(es);
     }
     catch(Exception e)
     {
       System.out.println(e);
     }
   }
}
```

The output of the program will be similar to

Enter Credit limit increment: 1000

 Procedure Executed

1	CHARLIE AND CO	20	47200.0
2	ARVIND MILLS	30	902200.0
3	PANDY BROTHERS	20	14700.0

5.4 Other JDBC Classes

Now those have seen all of the main database-related classes, look at some of the supporting classes that are available in JDBC. These classes include the Date, Time, TimeStamp, and so on. Most of these classes extend the basic Java classes to add capability to handle and translate data types that are specific to SQL.

java.sql.Date

This package (see Table 4.10) gives a Java program the capability to handle SQL DATE information with only year, month, and day values.

Return Type	Method Name	Parameter
Date	Date	(int year, int month, int day)
Date	Date	(long date)
Date	valueOf	(String s)
String	toString	()
int	getHours	()
int	getMinutes	()
int	getSeconds	()
void	setHours	(int Hr)
void	setMinutes	(int Min)
void	setSeconds	(int Sec)
void	setTime	(long date)

Table 4.10 java.sql.Date Methods

java.sql.Time

As seen in Table 4.11, the java.sql.Time adds the Time object to the java.util.Date package to handle only hours, minutes, and seconds. java.sql.Time is also used to represent SQL TIME information.

Return Type	Method Name	Parameter
Time	Time	(int hour, int minute, int second)
Time	Time	(long time)
Time	Time	valueOf(String s)
String	toString	()
Int	getDate	()
int	getDay	()
int	getMonth	()
int	getYear	()
void	setDate	(int date)
void	setMonth	(int month)
void	setTime	(int time)
void	SetYear	(int year)

Table 4.11 java.sql.Time Methods

java.sql.Timestamp

The java.sql.Timestamp package adds the TimeStamp class to the java.util.Date package (see Table 4.12). It adds the capability of handling nanoseconds. But the granularity of the subsecond timestamp depends on the database field as well as the operating system.

Return Type	Method Name	Parameter
TimeStamp	TimeStamp	(int year, int month, int date, int hour, int minute, int second, int nano)

TimeStamp	TimeStamp	(long time)
TimeStamp	valueOf	(String s)
String	toString	()
int	getNanos	()
void	setNanos	(int n)
boolean	after	(TimeStamp ts)
boolean	before	(TimeStamp ts)
boolean	equals	(TimeStamp ts)

Table 4.12 java.sql.Timestamp Methods

java.sql.Types

This class defines a set of XOPEN equivalent integer constants that identify SQL types. The constants are final types. Therefore, they cannot be redefined in applications or applets. Table 4.13 lists the constant names and their values.

Constant Name	Value
BIGINT	-5
BINARY	-2
BIT	-7
CHAR	1
DATE	91
DECIMAL	3
DOUBLE	8
FLOAT	6
INTEGER	4
LONGVARBINARY	-4
LONGVARCHAR	-1
NULL	0
NUMERIC	2
OTHER	1111
REAL	7
SMALLINT	5

TIME	92
TIMESTAMP	93
TINYINT	-6
VARBINARY	-3
VARCHAR	12

Table 4.13 java.sql.Types Constants

Moving the Cursor in Scrollable Result Sets

One of the new features in the JDBC 2.0 API is the ability to move a result set's cursor backward as well as forward. There are also methods that are used to move the cursor to a particular row and check the position of the cursor. Scrollable result sets make it possible to create a GUI (graphical user interface) tool for browsing result sets, which will probably be one of the main uses for this feature. Another use is moving to a row in order to update it.

Before taking advantage of these features, it is easy to create a scrollable ResultSet object. The following line of code illustrates one way to create a scrollable ResultSet object:

Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,

ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT * FROM CUSTOMERS ");

This code is similar to the earlier ResultSet object, except that it adds two arguments to the method createStatement. The first argument is one of three constants added to the ResultSet API to indicate the type of a ResultSet object:

TYPE_FORWARD_ONLY ,
TYPE_SCROLL_INSENSITIVE , and
TYPE_SCROLL_SENSITIVE .

The second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable: CONCUR_READ_ONLY and CONCUR_UPDATABLE. The point to remember here is that if a type is specified, it must also be specified whether it is read-only or

updatable. Also, first of all the type must be specified, because both parameters are of type int , the compiler will not complain if the order is switched.

Specifying the constant TYPE_FORWARD_ONLY creates a nonscrollable result set, that is, one in which the cursor moves only forward. If the type is not specified the default is TYPE_FORWARD_ONLY and CONCUR_READ_ONLY (as is the case of using only the JDBC 1.0 API).

Scrollable ResultSet object can be got only by specifying the following ResultSet constants:

TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE.

The difference between the two has to do with whether a result set reflects changes that are made to it while it is open and whether certain methods can be called to detect these changes. Generally speaking, a result set that is TYPE_SCROLL_INSENSITIVE does not reflect changes made while it is still open and one that is TYPE_SCROLL_SENSITIVE does. All three types of result sets will make changes visible if they are closed and then reopened. At this stage, not need to worry about the finer points of a ResultSet object's capabilities. No matter what type of result set to specify, limitation is that DBMS and driver actually provide.

A scrollable ResultSet object can be used to move the cursor around in the result set. Remember that when a new ResultSet object is created, it had a cursor positioned before the first row. Even when a result set is scrollable, the cursor is initially positioned before the first row. In the JDBC 1.0 API, the only way to move the cursor was to call the method next . This is still the appropriate method to call when it is absolutely necessary to access each row once, going from the first row to the last row, but there are so many other ways to move the cursor.

The counterpart to the method next, which moves the cursor forward one row (toward the end of the result set), is the new method **previous**, which moves the cursor backward (one row toward the beginning of the result set). Both methods return false when the cursor goes beyond the result set (to the position after the last row or before the first row), which makes it possible to use them in a while loop. The next method has been already used in the while loop.

have already used the method next in a while loop, but to refresh the memory, here is an example in which the cursor moves to the first row and then to the next row each time it goes through the while loop. The loop ends when the cursor has gone after the last row, causing the method next to return false. The following code fragment prints out the values in each row of srs, with five spaces between the name and price:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(
    "SELECT * FROM CUSTOMERS");
while (srs.next())
{
        int no=rs.getInt(1);
          String company=rs.getString(2);
          int rep=rs.getInt(3);
          double credit=rs.getDouble(4);
        System.out.println(no+"\t\t"+company+"\t"+rep+"\t\t"+credit);

}
```

The printout will look something like this:

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
1	CHARLIE AND CO	20	47200
2	ARVIND MILLS	30	902200
3	PANDY BROTHERS	20	14700

All of the rows is ResultSet going backward, but to do this, the cursor must start out being after the last row. Move the cursor explicitly to the position after the last row with the method **aftertaste** . Then the method previous moves the cursor from the position after the last row to the last row, and then to the previous row with each iteration through the while loop. The loop ends when the cursor reaches the position before the first row, where the method previous returns false.

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
```

```
ResultSet srs = stmt.executeQuery("SELECT * FROM CUSTOMERS");
srs.afterLast();
while (srs.previous()) {
        int no=rs.getInt(1);
          String company=rs.getString(2);
          int rep=rs.getInt(3);
          double credit=rs.getDouble(4);
    System.out.println(no+"\t\t"+company+"\t"+rep+"\t\t"+credit);
}
```

The printout will look similar to this:

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
1	PANDY BROTHERS	20	14700
2	ARVIND MILLS	30	902200
3	CHARLIE AND CO	20	47200

Moving the cursor to a particular row in a ResultSet object. The methods **first, last, beforeFirst,** and **afterLast** move the cursor to the row indicated in their names. The method **absolute** will move the cursor to the row number indicated in the argument passed to it. If the number is positive, the cursor moves the given number from the beginning, so calling absolute(1) puts the cursor on the first row. If the number is negative, the cursor moves the given number from the end, so calling absolute(-1) puts the cursor on the last row. The following line of code moves the cursor to the fourth row of srs :
srs.absolute(2);

If srs has 500 rows, the following line of code will move the cursor to row 497:
srs.absolute(-4);

Three methods move the cursor to a position relative to its current position. The method next moves the cursor forward one row, and the method previous moves the cursor backward one row. With the method relative how many rows to move from the current row and also the direction in which to move can be specified. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows. For example, in the following code fragment, the cursor moves to the fourth row, then to the first row, and finally to the third row:

srs.absolute(4); // cursor is on the fourth row

```
...
```

srs.relative(-3); // cursor is on the first row

```
...
```

srs.relative(2); // cursor is on the third row

The method getRow lets check the number of the row where the cursor is positioned. For example, use getRow to verify the current position of the cursor in the previous example as follows:

srs.absolute(4);

int rowNum = srs.getRow(); // rowNum should be 4

srs.relative(-3);

int rowNum = srs.getRow(); // rowNum should be 1

srs.relative(2);

int rowNum = srs.getRow(); // rowNum should be 3

Four additional methods verify whether the cursor is at a particular position. The position is stated in their names: isFirst, isLast, isBeforeFirst, isAfterLast. These methods all return a boolean and can therefore be used in a conditional statement. For example, the following code fragment tests to see whether the cursor is after the last row before invoking the method previous in a while loop. If the method isAfterLast returns false, the cursor is not after the last row, so the method afterLast is invoked. This guarantees that the cursor will be after the last row and that using the method previous in the while loop will cover every row in srs .

```
if (srs.isAfterLast() == false) {
      srs.afterLast();
}
while (srs.previous())
{
        int no=rs.getInt(1);
          String company=rs.getString(2);
          int rep=rs.getInt(3);
          double credit=rs.getDouble(4);
    System.out.println(no+"\t\t"+company+"\t"+rep+"\t\t"+credit);

}
```

Making Updates to Updatable Result Sets

Another new feature in the JDBC 2.0 API is the ability to update rows in a result set using methods in the Java programming language rather than having to send an SQL command. But before taking the advantage of this capability, create a ResultSet object that is updatable. In order to do this, supply the ResultSet constant CONCUR_UPDATABLE to the createStatement method. The Statement object it creates will produce an updatable ResultSet object each time it executes a query. The following code fragment illustrates creating the updatable ResultSet object uprs . Note that the code also makes uprs scrollable. An updatable ResultSet object does not necessarily have to be scrollable, but when making changes to a result set, generally want to be able to move around in it. A scrollable result set, can move to rows that need to be changed, and if the type is TYPE_SCROLL_SENSITIVE, it can get the new value in a row after the chang had made.

```
Connection con = DriverManager.getConnection("jdbc:odbc:oraodbc","scott","tiger");
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT * FROM CUSTOMERS ");
```

The ResultSet object uprs might look something like this:

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
1	CHARLIE AND CO	20	47200
2	ARVIND MILLS	30	902200
3	PANDY BROTHERS	20	14700

We can now use the new JDBC 2.0 methods in the ResultSet interface to insert a new row into uprs , delete an existing row from uprs , or modify a column value in uprs .

Updating a Result Set Programmatically

An update is the modification of a column value in the current row. Using the JDBC 1.0 API, the update would look something like this:

```
stmt.executeUpdate("UPDATE  CUSTOMERS  SET  CREDIT_LIMIT  =  90100"  +
"WHERE CUST_REP = 20");
```

The following code fragment shows another way to accomplish the update, this time using the JDBC 2.0 API:

uprs.last();
uprs.updateFloat("CREDIT_LILMIT", 90100f);

Update operations in the JDBC 2.0 API affect column values in the row where the cursor is positioned, so in the first line the ResultSet uprs calls the method last to move its cursor to the last row (the row where the column CREDIT_LIMIT has the value 20). Once the cursor is on the last row, all of the update methods will operate on that row until the cursor is moved to another row. The second line changes the value in the CREDIT_LIMIT column to 90100 by calling the method updateFloat. This method is used because the column value we want to update is a float in the Java programming language.

The Resultet. updateXXX methods take two parameters: the column to update and the new value to put in that column. As with the ResultSet. getXXX methods, the parameter designating the column may be either the column name or the column number. There is a different updateXXX method for updating each datatype (updateString , updateBigDecimal , updateInt , and so on) just as there are different getXXX methods for retrieving different datatypes.

uprs.last();

uprs.updateFloat("CREDIT_LIMIT", 90100f);

uprs.updateRow();

If the cursor is moved to a different row before calling the method updateRow , the update would have been lost. If, on the other hand, that the CREDIT_LIMIT should really have not been changed for Charlie and co it is easy to cancel the update by calling the method cancelRowUpdates . To invoke cancelRowUpdates before invoking the method updateRow ; once updateRow is called, calling the method cancelRowUpdates does nothing. Note that cancelRowUpdates cancels all of the updates in a row, so if there are many invocations of the

updateXXX methods on the same row, just charlie and co cannot be canceled. The following code fragment first cancels updating the CREDIT_LIMIT to 90100.

uprs.last();

uprs.updateFloat("CREDIT_LIMIT", 90100);

uprs.cancelRowUpdates();

In this example, only one column value was updated, but calling an appropriate updateXXX method for any or all of the column values in a single row. The concept to remember is that updates and related operations apply to the row where the cursor is positioned. Even if there are many calls to updateXXX methods, it takes only one call to the method updateRow to update the database with all of the changes made in the current row.

Inserting and Deleting Rows Programmatically

In the previous section, we saw how to modify a column value using methods in the JDBC 2.0 API rather than having to use SQL commands. With the JDBC 2.0 API, see also inserting a new row into a table or deleting an existing row programmatically.

Let's suppose that our coffee house proprietor is getting a new variety from one of his coffee suppliers, The High Ground, and wants to add the new coffee to his database. Using the JDBC 1.0 API, he would write code that passes an SQL insert statement to the DBMS. The following code fragment, in which stmt is a Statement object, shows this approach:

```
stmt.executeUpdate("INSERT INTO CUSTOMERS " +
      "VALUES (4,'PRAKASH POLYMERS', 40, 35000)");
```

The same thing can be done without using any SQL commands by using ResultSet methods in the JDBC 2.0 API. Basically, after a ResultSet object with results from the table CUSTOMERS, can build the new row and then insert it into both the result set and the table CUSTOMERS in one step. Build a new row in what is called the insert row, a special row associated with every ResultSet object. This row is not actually part of the result set.

First step will be to move the cursor to the insert row, which can be done by invoking the method moveToInsertRow . The next step is to set a value for each column in the row. This can be done by calling the appropriate updateXXX method for each value. Note that these are the same updateXXX methods used in the previous section for changing a column value. Finally, call the method insertRow to insert the row that just populated with values into the result set. This one method simultaneously inserts the row into both the ResultSet object and the database table from which the result set was selected.

The following code fragment creates the scrollable and updatable ResultSet object uprs , which contains all of the rows and columns in the table COFFEES :

```
Connection con =
DriverManager.getConnection("jdbc:odbc:oraodbc","scott","tiger");
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                              ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT * FROM
Customers");
```

The next code fragment uses the ResultSet object uprs to insert the row for PRAKASH POLYMERS, shown in the SQL code example. It moves the cursor to the insert row, sets the five column values, and inserts the new row into uprs and CUSTOMERS :

```
uprs.moveToInsertRow();
uprs.updateInt("CUST_NUM", 4);
uprs.updateString("COMPANY", "PRAKASH POLYMERS");
uprs.updateInt("CUST_REP", 40);
uprs.updateFloat("CREDIT_LIMIT", 35000);
uprs.insertRow();
```

Because either the column name or the column number to indicate the column to be set, code for setting the column values could also have looked like this:

```
uprs.updateInt(1, 4);
uprs.updateString(2, "PRAKASH POLYMERS");
uprs.updateInt(3, 40);
uprs.updateFloat(4, 35000);
```

To insert a row do not supply a value for every column in the row. If a value for a column is omitted that was defined to accept SQL NULL values, then the value assigned to that column is NULL. If a column does not accept null values, however, a SQLException when an updateXXX method is not called, to set a value for it. This is also true if a table column is missing in the ResultSet object.

In the example above, the query was SELECT * FROM CUSTOMERS, which produced a result set with all the columns of all the rows. To insert one or more rows, the query does not have to select all rows, but it is safer to select all columns. Especially if the table has hundreds or thousands of rows, use a WHERE clause to limit the number of rows returned by the SELECT statement.

After having called the method insertRow, just start building another row to be inserted, or move the cursor back to a result set row. For instance, invoke any of the methods that put the cursor on a specific row, such as first , last , beforeFirst , afterLast , and absolute .   Also use the methods previous , relative , and moveToCurrentRow . Note that moveToCurrentRow can only be invoked when the cursor is on the insert row.

To call the method moveToInsertRow , the result set records which row the cursor is sitting on, which is by definition the current row. As a consequence, the method moveToCurrentRow can move the cursor from the insert row back to the row that was previously the current row.

Code Sample for Inserting a Row

The following code sample is a complete program that should run if have a JDBC 2.0 Compliant driver that implements scrollable result sets. At the time of this writing there are not yet any JDBC 2.0 Compliant drivers, so this code, though it compiles, has not been tested on a driver and DBMS.

The ResultSet object uprs is updatable, scrollable, and sensitive to changes made by itself and others. Even though it is TYPE_SCROLL_SENSITIVE, it is possible that the getXXX methods called after the insertions will not retrieve values for the newly-inserted rows. There are methods in the DatabaseMetaData interface that will tell what is visible and what is detected

in the different types of result sets for the driver and DBMS. These methods are discussed in detail in JDBC Database Access with Java, but they are beyond the scope of this tutorial. In this code sample we wanted to demonstrate cursor movement in the same ResultSet object, so after moving to the insert row and inserting two rows, the code moves the cursor back to the result set, going to the position before the first row. This puts the cursor in position to iterate through the entire result set using the method next in a while loop. To be absolutely sure that the getXXX methods include the inserted row values no matter what driver and DBMS is used, close the result set and create another one, reusing the same Statement object stmt and again using the query SELECT * FROM CUSTOMERS.

After all the values for a row have been set with updateXXX methods, the code inserts the row into the result set and the database with the method insertRow. Then, still staying on the insert row, it sets the values for another row.

```java
import java.sql.*;
import java.io.*;

class ScrollSelect
{
    public static void main(String[] args)
    {

        try
        {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection
 ("jdbc:odbc:oraodbc","scott","tiger");
            Statement stmt = con.createStatement
                      (ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT * FROM
                CUSTOMERS");
            rs.moveToInsertRow();
            rs.updateInt("CUST_NO", 4);
 rs.updateString("COMPANY_NAME", "PRAKASH POLYMERS");
            rs.updateInt("CUST_REP", 40);
            rs.updateFloat("CREDIT_LIMIT", 35000f);
            rs.insertRow();
```

```
            rs.updateInt("CUST_NO", 5);
             rs.updateString("COMPANY_NAME", "BALAJI
      BROTHERS");
              rs.updateInt("CUST_REP", 50);
              rs.updateFloat("CREDIT_LIMIT", 50000f);
            rs.insertRow();

            rs.beforeFirst();

System.out.println("Table CUSTOMERS after insertion:");
       System.out.println("CUST_NUM" + "\tCOMPANY" +
      "\t\tCUST_REP" + "\tCREDIT_LIMIT");

              while (rs.next())
              {
                    int no=rs.getInt(1);
                    String company=rs.getString(2);
                     int rep=rs.getInt(3);
                     double credit=rs.getDouble(4);

      System.out.println(no+"\t\t"+company+
                            "\t"+rep+"\t\t"+credit);

              }
              rs.close();
              stmt.close();
              con.close();
System.out.println("Records succesfully selected");
       }catch(Exception e)
       {
              e.printStackTrace();
       }
    }
}
```

Deleting a Row

Deleting a row is the third way to modify a ResultSet object, and it is the simplest. To do is move the cursor to the row that need to be deleted and then call the method deleteRow . For example, to delete the fourth row in the ResultSet uprs , the code will look like this:

uprs.absolute(4);

uprs.deleteRow();

The fourth row has been removed from uprs and also from the database.

The only issue about deletions is what the ResultSet object actually does when it deletes a row. With some JDBC drivers, a deleted row is removed and is no longer visible in a result set. Some JDBC drivers use a blank row as a placeholder (a "hole") where the deleted row used to be. If there is a blank row in place of the deleted row, use the method absolute with the original row positions to move the cursor because the row numbers in the result set are not changed by the deletion.

In any case, remember that JDBC drivers handle deletions differently. For example, to write an application meant to run with different databases, don't write code that depends on there being a hole in a result set.

Seeing Changes in Result Sets

To modify data in a ResultSet object, the change will always be visible if the object is closed, and then reopen it. In other words, if the same query is re-executed, it will produce a new result set, based on the data currently in a table. This result set will naturally reflect changes anyone made earlier.

With a ResultSet object that is TYPE_SCROLL_SENSITIVE, seeing the updates anyone makes to column values. The only way to be sure is to use DatabaseMetaData methods that return this information.

To some extent regulate what changes are visible by raising or lowering the transaction isolation level for the connection with the database. For example, the following line of code, where con is an active Connection object, sets the connection's isolation level to TRANSACTION_READ_COMMITTED:

con.setTransactionIsolation(TRANSACTION_READ_COMMITTED);

With this isolation level, the ResultSet object will not show any changes before they are committed, but it can show changes that may have other consistency problems. To allow fewer data inconsistencies, raise the transaction isolation level to TRANSACTION_REPEATABLE_READ. The problem is that the higher the isolation level, the poorer the performance. And, as is always true of databases and drivers, the limitation is what they actually provide. Many programmers just use their database's default transaction isolation level.

In a ResultSet object that is TYPE_SCROLL_INSENSITIVE, generally cannot see changes made to it while it is still open. Use only this type of ResultSet object because they want a consistent view of data and do not want to see changes made by others.

Use the method refreshRow to get the latest values for a row straight from the database. This method can be very expensive, especially if the DBMS returns multiple rows each time call refreshRow . Nevertheless, its use can be valuable if it is critical to have the latest data. Even when a result set is sensitive and changes are visible, an application may not always see the very latest changes that have been made to a row if the driver retrieves several rows at a time and caches them. Thus, using the method refreshRow is the only way to be sure that you are seeing the most up-to-date data.

The following code sample illustrates how an application might use the method refreshRow when it is absolutely critical to see the most current values. Note that the result set should be sensitive; that is the method refreshRow with a ResultSet object that is TYPE_SCROLL_INSENSITIVE , refreshRow does nothing.

```
Statement stmt = con.createStatement(
                        ResultSet.TYPE_SCROLL_SENSITIVE,
     ResultSet.CONCUR_UPDATABLE);
     ResultSet rs=stmt.executeQuery("SELECT * from
     customers ");
     rs.absolute(4);

float AMOUNT1 = rs.getFloat("CDREDIT_LIMIT");

// do something. . .
rs.absolute(4);
rs.refreshRow();
float AMOUNT2 = rs.getFloat("CREDIT_LIMIT");
if (AMOUNT1>AMOUNT2) {
```

```
        // do something. . .
}
```

5.5 Short Summary

- There are 3 more drivers supplied with Sun Java and so many vendors coming out with more drivers.
- Deleting a row is the third way to modify a ResultSet object, and it is the simplest.
- JDBC 2.0 API can able to update rows in a result set using methods in the Java programming language rather than having to send an SQL command.

5.6 Brain Storm

1. Write a note on updating a Resultset programmatically.

2. Explain Prepared statements and Callable statements.

3. Explain varies JDBC classes.

�…ဩ

Lecture 6

Javabeans

Objectives

In this lecture you will learn the following:

- ✎ Introduction to Software Component Models

- ✎ Introduction to JavaBean

- ✎ Working with Beans Development Kit(BDK)

```
┌─────────────────────────────────────────────────┐
│ ┌───────────────────────────────────────────────┐ │
│ │                Coverage Plan                    │ │
│ └───────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────┘
```

Lecture 6

6.1 Snap shot

This chapter provides an overview of the Software Component Models, features of a software component and an overview of the JavaBeans Technology. Java promises a truly open cross-platform execution environment for permitting the creation of applications once with deployment everywhere. So far, this "write once, use everywhere" technology has been restricted to programmers. However, the Java component model, JavaBeans, will provide these benefits too real people, the non-programmers of the world, finally fulfilling the promise of software component technology.

6.2 Introduction to Software Components

Software components are self-contained, reusable software units. Software development with a component object model is like building with Legos building blocks. Instead of building an entire application from scratch, you build an application by hooking together your existing building blocks. This approach not only saves time, money, and effort, but it produces more consistent, reliable applications.

Software components raise the level of software reuse, moving its emphasis from the level of the skilled programmer to the level of the business application creator. Note this term "business application creator." What it means is that software component can be assembled into applications by people skilled in the business, but almost certainly not skilled in programming.

6.2.1 Need For Software components

Think of a hi-fi system of separate components. Skilled electronics engineers build each component: amplifier, CD player, speakers, etc., from small parts (integrated circuits, wires, circuit boards, etc.). On the amplifier are user controls, on/off switches, volume control, etc., and, at the back, connections. The connections use different types of plugs for different functions to stop me from plugging the speakers into the wrong CD input socket. As the user, one is expected to be able to understand the controls and to work out which cable plugs in where. He need not know how the electronic circuitry operates.

There are numerous similarities between a hi-fi system and the ideal of computer software built from components. Consider an HTML page, to be delivered across the World Wide

Web, which allows me to select an item from a catalogue and to buy it using a credit card. The page must contain a way for me to enter my credit card details, a way to indicate my product choice, and a way to allow me to say "buy." In this ideal world, the page is created by somebody skilled in publishing on the Web, and not by a programmer. The publisher builds up the Web page and includes on it three software components. One that takes credit card details, one that allows me to select an item, and one that displays a button with the word "buy" on it.

The publisher "wires" the components together by connecting the "credit card details available" event from the credit card component and the "product selection" event from the selector component to the "buy" component. Now, when the user has entered his or her credit card details and selected an item and clicked on the "buy" button, a request is sent to the Web server. The constructor of the HTML page knows nothing of how the credit card component verifies its details or of how the transaction component processes the request; the HTML author simply plugs the pieces together to provide the desired end result: a functional Web page.

6.2.2 Classifications of software components

Software components can be broadly classified into visual and non-visual components. Non-Visual Components: An alarm is an example of a typical non-graphical component. When you add it to an application, you can graphically choose how frequently the alarm goes off and can easily edit the alarm code. But when the applications runs, alarm component has no visual appearance.

Visual Components: Developing software with a component object model allows   to purchase or develop software components and then integrating them in a graphical environment(GUI builder tool) to create a complete application.Reusable software components can be simple, such as buttons, text fields, listboxes, scrollbars, and dialogs.eg: Font selectors, Database viewers etc.,

The following is a slider component created in Java. This slider can be integrated with any other components to get a complete application.
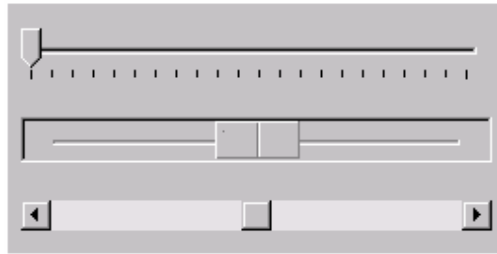
Figure 1.1 Slider created in Java

The above components are traditionally sold by third parties. More recently, vendors sell more complex software components, such as calendars and spreadsheets.

6.3 Software Component Model

A simple definition: *A software component model is a specification for how to develop reusable software components and how these component objects can communicate with each other.*

We can now focus on how component models work in a general sense. *Component developers* must deal with the low-level details of implementing a component and its associated classes. *Application developers* can then simply hook together existing components in a visual development environment.This is what we call rapid application development (RAD)-- existing codebase can be easily reused (or purchased from an independent vendor) and integrated in a third-party development environment.

6.3.1Features of Software Component

Understanding the Software Components themselves in order for a software component model to work, each software component must provide several features:

- *The component must be able to describe itself*. This means that a component must be able to identify any properties that can be modified during its configuration and also the events that it generates. This information is used by the development environment to seamlessly integrate third-party components.

- *The component must allow graphical editing of its properties*. A RAD environment is intensely graphical; the configuration of a software component is done almost exclusively through control panels that expose accessible properties.

- *The component must be directly customizable from a programming language.* Because components also can be connected by scripting languages or other environments that can only access the components from a code level, this feature also allows software components to be manipulated in non-visual development projects.

- *The component must generate events* or provide some other mechanism that lets programmers semantically link components. This means that the programmer can easily add the appropriate action code to buttons so that button clicks will properly affect other components.

6.4 Javabean

A JavaBean is a reusable software component that is written in Java programming language. It can be visually manipulated in builder tools. A JavaBean is often referred to simply as a Bean.

6.4.1 Importance of Java Component Model

Cross platform deployment: The promise of the Java environment is the software producer's nirvana: write it once and execute it anywhere. The platform-independent nature of Java and its standardized class library finally make it possible to develop a single version of an application and have it execute on any Java-compatible system without the need for recompilation or addition of special logic. Components created with existing component models, such as Microsoft's ActiveX and OpenDoc, do not meet the needs of cross-platform deployment.

Secured: OpenDoc is available on Apple Macintosh, OS/2, and Windows, but again, one component executable does not fit all! Existing components are installed and registered as part of the operating system. If this model is extended to the World Wide Web, the security implications are extremely scary! An ActiveX or OpenDoc component loaded dynamically as part of a Web page has access to the full range of operating system interfaces and, therefore, once installed and registered, may do whatever it pleases with the user's system. A Java

component loaded from the Web executes within the Java "padded cell" or "sandbox," where its access to the user's system can be strictly controlled and where, by default, it can do no damage.

6.4.2 JavaBeans Objectives

Given the failings of the existing component models, some of the aims of JavaBeans are fairly obvious. Here are a few of the targeted characteristics:

- *Portable*: Written in Java with no platform-native code.

- *Lightweight*: It should be possible to implement a component as small as a push button or as large as a complete spreadsheet or word processor.

- *Simple to create*: It should be a simple job to create a Java component without implementing countless methods. Creation should be possible with or without development tools. It should be simple to migrate from a simple applet to a Java component.

- *Hostable in other component models*: It should be possible to use a JavaBean as a first-class ActiveX, OpenDoc, or other component. A JavaBean may be contained within an ActiveX or OpenDoc container, such as a word processor, and will behave exactly as if it were a native component. Thus a JavaBean chart component can be embedded in a word processor document to interact with a spreadsheet in the same document. The creator of the JavaBean need not provide special logic to deal with being hosted in another environment, and, indeed, the bean may not even know it is happening, because all communication and conversion is provided by a "bridge."

- *Able to access remote data*: A Java component may use any of the standard distributed object (JavaIDL or Remote Method Invocation) or distributed data (JDBC) mechanisms to access remote data. In fact, a bean may use any of the standard environment facilities.

6.4.3 Basic Bean Concepts

JavaBeans, regardless of their functionality, are defined by the following features.

- *Introspection* — Beans support introspection, which allows a builder tool to analyze how Beans work. They adhere to specific rules called design patterns for naming Bean features. Each Bean has a related Bean information class, which provides property, method, and event information about the Bean itself. Each Bean information class implements a BeanInfo interface, which explicitly lists the Bean features that are to be exposed to application builder tools.

- *Properties* — Properties control a Bean's appearance and behavior. Builder tools introspect on a Bean to discover its properties and to expose them for manipulation. As a result, you can change a Bean's property at design time.

- *Customization* — The exposed properties of a Bean can be customized at design time. Customization allows a user to alter the appearance and behavior of a Bean. Beans support customization by using property editors or by using special, sophisticated Bean customizers.

- *Events* — Beans use events to communicate with other Beans. Beans may fire events, which means the Bean sends an event to another Bean. When a Bean fires an event it is considered a source Bean. A Bean may receive an event, in which case it is considered a listener Bean. A listener Bean registers its interest in the event with the source Bean. Builder tools use introspection to determine those events that a Bean sends and those events that it receives.

- *Persistence* — Beans use Java object serialization, implementing the java.io.Serializable interface, to save and restore state that may have changed as a result of customization. State is saved, for example, when we customize a Bean in an application builder, so that the changed properties can be restored at a later time.

- *Methods* — All JavaBean methods are identical to methods of other Java classes. Bean methods can be called by other Beans or via scripting languages. A JavaBean public method is exported by default.

> **Note:** While Beans are intended to be used primarily with builder tools, they need not be. Beans can be manually manipulated by text tools through programmatic interfaces. All key APIs, including support for events, properties, and persistence, are designed to be easily read and understood by programmers, as well as by builder tools.

6.5 Bean Development Kit

The Bean Development kit is a tool that allows one to configure and interconnect a set of Beans. Using it, you can change the properties of a Bean, link two or more Beans, and watch Beans Execute.Therefore the BDK provides an easy way for you to test Beans that you write and to explore the Capabilities of Beans written by others. The BDK also includes a set of demonstration Components and their source code. The BDK is a java application.

For creating the first JavaBean, let's look at the BDK BeanBox. We can create a JavaBean and then use the BeanBox to test that it runs properly. If a JavaBean runs properly in the BeanBox, it is sure that it works properly with other commercial builder tools.

6.5.1 Starting the BeanBox

Type **cd** *root_directory***:\bdk1.1\beanbox** (say c:/bdk1.1/beanbox) to get to the appropriate directory.

Then, type *run* to start the BDK.

When the BeanBox is started, we can see three windows:

- ToolBox window
- BeanBox window
- Properties window

The ToolBox window displays the JavaBeans that are currently installed in the BeanBox, such as the Beans the come with the BeanBox demo. When the BeanBox starts, it automatically loads its ToolBox with the Beans in the JAR files contained in the bean/jars directory. We can add additional Beans, such as our own Beans, to the ToolBox.

The BeanBox window itself appears initially as an empty window. We use this empty window for building applications.

The Properties window displays the current properties for the selected Bean. If no Bean is selected, such as when we first start the BeanBox or if we click in the BeanBox window's

background, then the Properties window displays the BeanBox properties. We can use the Properties window or sheet to edit a Bean's properties.

Using the BDK BeanBox and the Demo JavaBeans

The easiest way to understand how the BeanBox works is to use it. The BeanBox enables us to construct simple Beans applications without writing any Java code.

Example: Juggling Duke

As a first example, we will build a simple "Juggling Duke" application in which Duke will start or stop juggling depending on which of two buttons you push. Follow the steps given below in the same order.

STEP 1: Click on Juggler Bean to select from the list of Beans in the Toolbox window and notice the cursor change.

STEP 2: Place the cursor anywhere in the BeanBox, then click the mouse. This inserts a Juggler Bean into the BeanBox window. The highlighted box surrounding the Juggler indicates the Juggler is the currently selected bean.
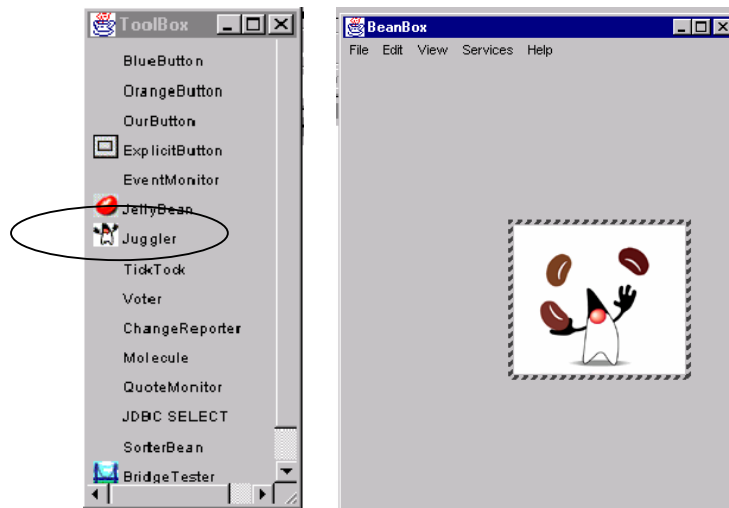
Figure 1.2 Demonstration Juggler Bean

STEP 3: Next we'll look at adding a start button to control the Juggler. This button Bean is an instance of the OurButton Bean class. Click the OurButton Bean name in the ToolBox, then place an instance of the button in the BeanBox.

STEP 4: Select the button in the BeanBox so that the button's properties display in the Property sheet. Edit the label field in the button's Property sheet so that the button's label reads "start."

STEP 5: Use the BeanBox Edit menu to select an action event to be   fired by the start button. Before choosing the event action, be sure that you have selected the start button.

Notice that once you select the actionPerformed menu item, BeanBox enters a state where a line emanates from the start button and follows the mouse as you move it around the window. This indicates that the button is the selected source for the action event, and that your next mouse press should be over the target Bean, which defines appropriate event-handler methods, in this case the Juggler Bean.

STEP 5: Drag the line from the start button and release it over the Juggler Bean. A dialog appears listing applicable event handlers defined by the Juggler Bean.

STEP 6: Select the startJuggling method as the target for the event, then press OK.

Now, when you press the start button Duke should start tossing beans around in a circle over his head like a professional juggler.

You can control Duke's juggling speed by manually setting the property value labeled animationRate in the Juggler's property sheet editor. For the appropriate property sheet editor to appear, the Juggler must be the currently selected Bean within the BeanBox frame.

To complete the example program, add a stop button. Repeat the steps you took when you added the start button and connected it to the appropriate Juggler action.

For the new button,

STEP 7: Edit the label field of the property sheet to read "stop."

STEP 8: Hook up the action event from the stop button to the Juggler's stopJuggling event-handler method. Make sure the stop button is the currently selected bean.

STEP 9: Select the actionPerformed event from the Edit/Events menu.

STEP 10: Drag the event line from the stop button source to the Juggler Bean target. Press and release the mouse button with the connection line over the Juggler.
The dialog of applicable event-handler methods defined for the Juggler Bean displays; select the stopJuggling event and click OK.
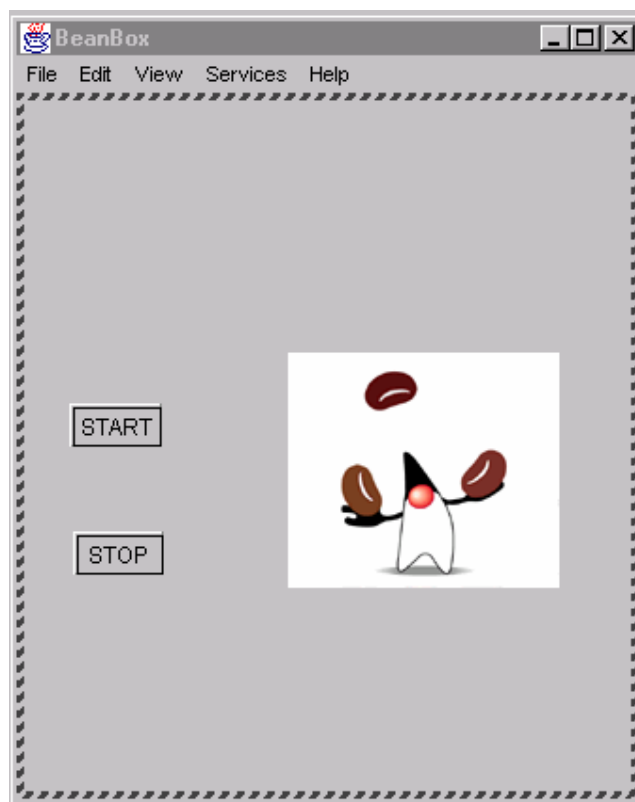

Fig Demonstration Beans bound with each other by Event Handling mechanism.

You should now be able to start and stop the juggler by pressing the appropriate button.

From the above example we have learnt,
• Droping Beans from the ToolBox into the BeanBox and changing their properties using the Properties sheet and associated property editors.   • One Bean can fire an event and another Bean can react to the fired event.

6.6 Short Summary

- A software component model is a specification for how to develop reusable software components and how these component objects can communicate with each other.

- Software components are self-contained, reusable software units Instead of building an entire application from scratch, you build an application by hooking together your existing building blocks.

- The Bean Development kit is a tool that allows one to configure and interconnect a set of Beans

- When the BeanBox is started, ToolBox, BeanBox and Properties Window will be displayed

- Software components are two types they are visual and non-visual components.

6.7 Brain Storm

1. What is a software component model?
2. What is the need for software component model?
3. Explain Classification of software components.
4. What are the features of software component model?
5. Write a short note on JavaBean.

ಬಿಃ

Lecture 7

Event Handling

Objectives

In this lecture you will learn the following

- ✍ About how to build java bean…

- ✍ About event handling

Coverage Plan

Lecture 7

7.1 Snap Shot

In this lecture we are going to discuss about how to built a Java bean and its working methodologies.

7.2 Building Simple Bean

7.2.1 Building the First Bean

Here we are going to construct a Bean called **Spectrum**. The component displays square 100 pixels wide and 100 pixels high, and fills it with colors of the spectrum. The component has one boolean property named *vertical*. If this property is true, the colors are arranged in a vertical direction. Otherwise, the colors are oriented in a horizontal direction.

Follow the instructions in the following sections to develop and test this Bean.

STEP 1: Create the source code

Create a directory named **spectrum** anywhere on your computer.

Create the source code

Enter the source code shown in the listing at the end of this paragraph. You must name this file Spectrum.java and place it in the spectrum directory. The package statement at the beginning of the file places this class in a package named spectrum. (Each example in this chapter is placed in its own package to avoid naming conflicts)

Listing Spectrum.java

```
import java.awt.*;
public class Spectrum extends Canvas  {

 private boolean vertical;

 public Spectrum() {
       vertical=true;
```

```
        setSize (100,100);
  }


  public boolean getVertical() {
        return (vertical);
  }


  public void setVertical(boolean vertical) {
        this.vertical=vertical;
        repaint ();
  }


  public void paint(Graphics g) {
        float saturation=1.0f;
        float brightness=1.0f;
        Dimension d=getSize ();
        if (vertical) {
            for (int y=0;y<d.height;y++) {
                float hue=(float) y/(d.height-1);
                g.setColor(Color.getHSBColor(hue,saturation,brightness));
                g.drawLine (0,y, d.width-1, y);
            }
        }
        else  {
            for (int x=0;x<d.width; x++)  {
                float hue=(float) x/(d.height-1);
                g.setColor(Color.getHSBColor(hue,
        saturation,brightness));
                g.drawLine (x, 0,x, d.width-1);
            }
        }
  }  // method ends
}// class ends
```

How the program works?

The above Spectrum class is a subclass of Canvas. The private boolean variable named vertical is one of its properties. The constructor initializes that property to true and sets the size of the component.

The method to access the property are **getVertical ()** and **setVertical ().** Note that when the property is changed via the setVertical () method, the repaint () method is invoked to update this display. The **paint ()** method fills the square with the colors of the spectrum. A specific color can be uniquely represented by its hue, saturation, and brightness. Each of these parameters is a float and ranges from 0.0 to 1.0f. In this code, the saturation and the brightness are set to 1.0f and the hue is varied from 0.0 to 1.0f. This is the mechanism used to compute the complete range of colors. The getSize () method is invoked to determine the dimensions of this Bean.

The vertical property is then checked to determine if the colors should change in the vertical or horizontal dimension. If the vertical is true, the square is filled by drawing horizontal lines of different colors. The hue of each line is calculated by scaling its y coordinate to a value between 0.0 to 1.0f. The getHSBColor () static method of the Color class accepts hue, saturation, and brightness parameters for a color and returns a reference to the Color object. That object is used to set the current color of the graphics context. Then a horizontal line of that color is drawn.

If the vertical is false, the square is filled by drawing vertical lines of different colors. The logic to do this is analogous to that described in the previous paragraph, except that the hue of each line is calculated by scaling its x coordinate to a value between 0.0 and 1.0f.

STEP 2: Compile the source code

Change to the parent directory of spectrum and type.,

javac spectrum\Spectrum.java

Check that the.class file has been created in the spectrum directory.

Note: Each code example in this Course material exists in a separate package. Java requires that the directory hierarchy mirror the package hierarchy. Therefore, we must always be in the parent directory when compiling a code example with **javac** or executing an application with **java.** Our CLASSPATH environment variable must include this parent directory. This is necessary so the .class files can be found.

STEP 3: Create the manifest template file

All Beans must be specified in a manifest template file. This file is used in the next step by the tool that packages your Bean into a JAR file. In this example, you must create a manifest template file in order to indicate that **Spectrum.class** is a Bean. The contents of this file always uses forward slashes in the path name of a file. Name the file **spectrum.mft** and place it in the spectrum directory.

```
Name: spectrum\Spectrum.class
Java-Bean:      True
```

Manifest files are examined in detail in the Appendix part of this Course material. Briefly, a manifest template file is the basis for a manifest file, which is the first element in a JAR file and describes its contents. In cases where you've to package several Beans into one JAR file, the manifest template file includes a separate entry for each Bean.

STEP 4: Create a JAR file

All Beans must be stored within a JAR file. Change to the parent directory of spectrum and enter the following command to create a JAR file containing the Spectrum Bean:
 jar cfm c:\bdk1.1\jars\spectrum.jar spectrum*.mft spectrum*.class

This command creates a jar file named spectrum.jar and places it in the C:\bdk\jars directory. This is the directory in which the BDK looks for JAR files. The manifest template file and the .class files in the spectrum directory are used. Later in this chapter we will examine JAR files in detail.

STEP 5: Start the BDK

Type cd c:\bdk1.1\beanbox to get to the appropriate directory. Then type run to start the BDK. You should see the Toolbox, Beanbox and Properties windows. Toolbox should have an entry labeled "Spectrum"

STEP 6: Test the Spectrum Bean

Create an instance of Spectrum in the BeanBox. You should see a square with the colors of the spectrum oriented in a vertical direction. Use the property window to change the vertical

property to false. Observe that the display changes immediately. You may create several instances of Spectrum in BeanBox. The vertical property for each can be changed independently.

The properties window also presents several other properties for this Bean (for example, background, foreground and font). These properties are defined by the Component class. Since our Bean is a subclass of Component, it also has these properties. However the values of these properties are not relevant to this Bean because it completely fills the square with colors and does not display text.
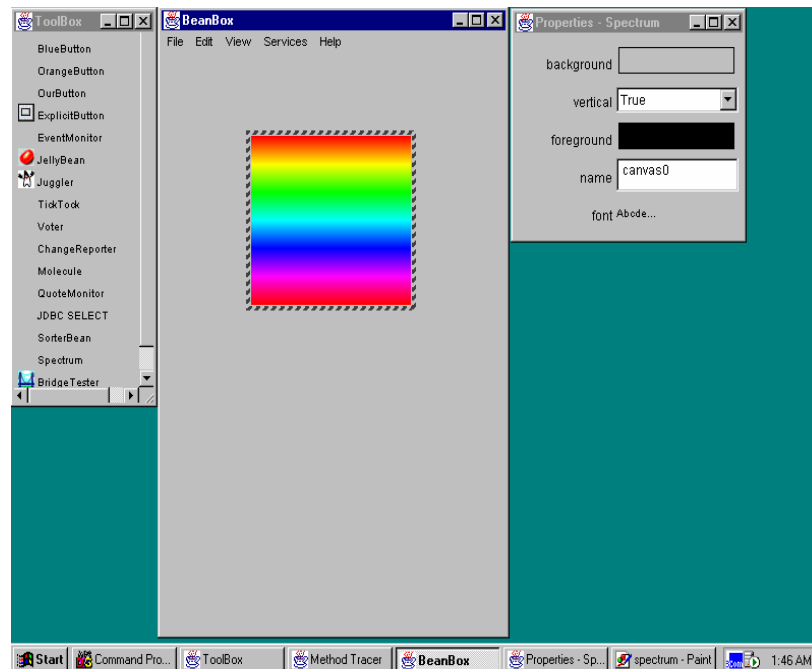

Figure 1.3 **creating an instance of Spectrum in the BeanBox**

In this example, the BDK used introspection to examine your new Bean and to automatically infer that vertical was one of its properties. This was possible because the access methods getVertical() and setVertical() followed a simple naming pattern. The property **vertical** is called Simple property because it has got only one get method **getVertical ()** and one set method **setVertical ().**

From the above example we have learnt,

- To create a bean of our own.
- To use getter and setter method to work on the property of a bean.

7.2 Event Handling

A Bean that wants to generate events needs to keep track of interested event targets. In the delegation event model, the event mechanism is broken up conceptually into *event dispatch* and *event handling*. Event dispatch is the responsibility of the event **source**; event handling is the responsibility of the event **listener**.

Any object that wants to know when an event is fired by a Bean can tell the Bean it wants to be informed about particular events. In other words, an event listener *registers* interest in an event by calling a predetermined method in the event source.
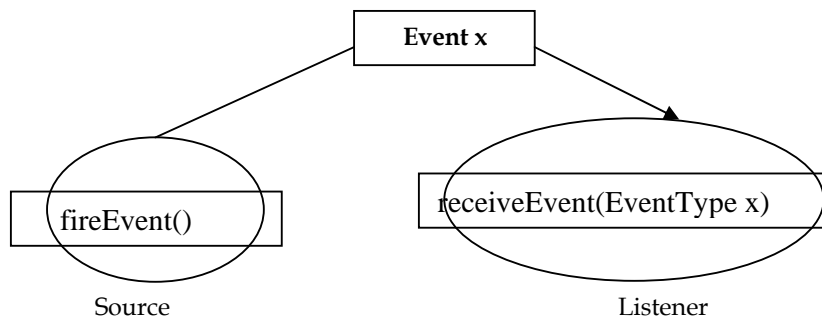
Source   Listener

7.2.1 Registering Event Listeners

Consider a typical button Bean that generates events when pressed. An interested listener Bean might increment a counter object each time the button is pressed.

If the button Bean wants to be an event source, it must provide two methods that can be called by interested objects.

One method adds the caller to the list of listeners who are notified when the event occurs.

```
public synchronized void addActionListener
    (ActionListener l) {...}
```

The other method removes the caller from the list of interested listeners.

```
public synchronized void removeActionListener
 (ActionListener l) {...}
```

7.2.2 Naming Event Listeners

Similar to properties, the signature of the method names must follow specific patterns. The Java introspection mechanism detects the pattern of the method's signature and can determine the events the source Bean generates from the name of the registration methods, together with the type of the arguments of the registration methods.

Java's introspection mechanism recognizes the following general pattern for event generation capabilities:

```
public synchronized
      void addTYPE(TYPE listener);
public synchronized
    void removeTYPE(TYPE listener);
```

Note that TYPE is replaced by the class name of the particular event listener; for example, MouseListener or MouseMotionListener.

7.2.3 Following the ActionEvent

When the above event registration methods are defined for our button Bean, Java's introspection mechanism is able to determine that an ActionEvent can be generated by the button. If the counter object wants to be notified when an ActionEvent occurs, it calls the button's addActionListener method, giving itself as an argument. For this to work, the counter object has to first implement the ActionListener interface, because the argument to addActionListener is an ActionListener object.

The button Bean needs to track the listeners who register to receive notification of ActionEvents. This is where the Vector import statement comes into play. The button Bean maintains a list (or Vector) of listeners. Thus, the source Bean declares the following line:

private Vector listeners = new Vector();

When the Bean's addActionListener is called, the listener supplied as an argument to the call is appended to the Vector of listeners, as follows:

```
public synchronized void addActionListener
    (ActionListener l) {
        listeners.addElement(l);
}
```

Similarly, when removeActionListener is called, the listener supplied as an argument to the method is removed from the list of listeners:

```
public synchronized void removeActionListener
    (ActionListener l) {
                listeners.removeElement(l);
}
```

Dispatching Events to Event Listeners

When an event is fired, the event source (the button Bean) iterates over the list of listeners and sends each listener a notification of the **ActionEvent.**

7.3 Short Summary

- Every Bean must have a zero-argument constructor. The builder tool creates an instance of the component by using that constructor.

- All Beans must be specified in a manifest template file.

- The Java introspection mechanism detects the pattern of the method's signature and can determine the events the source Bean generates…

- All Beans must be stored within a JAR file

7.4 Brain Storm

1. What are steps involved in creating JavaBean development.
2. Define Event handling
3. How can you naming Event Listners.

৲৩ড়

Lecture 8

Serialization and Deserialization

Objectives

In this lecture you will learn the following

- ✍ About Bean Persistence

- ✍ About Serialization and Deserialization

Coverage Plan

Lecture 8

8.1 Snap Shot

This lecture will discuss on what is bean presistence and what is serialization and deserialization how it is needed in java program.This section develops a Bean called graph that shows six nodes positioned at the corners of a hexagon. You can between nodes by moving the mouse to a node, pressing the mouse button, and dragging the mouse to another node and then releasing it.

8.2 Bean Persistence

Persistence - Saving and restoring Beans in the BDK
To better understand persistence, let's begin with an example:

1. Start the BDK and create an instance of the Spectrum Bean from Chapter 2. Its vertical property is initially true. Changes this value too false. You will see an immediate change in the apperance of the Bean.

2. Now select the file1 save menu options in BeanBox.A file dialog box titled "Save As" appears. Press the save button to serialize this Bean to the default file beanbox.temp.select the File Exit menu options to exit the BDK.

3. Start the BDK again. Select the file Load options in BeanBox.A file dialog box titled "Open" appears. Press the Open buttons to restore the Bean from the default file beanbox.tmp.Observe that the component appears, and its vertical property is false.

From the above example we have learnt,

- How a simple Bean with one property could be serialized and   deserialized.
- The essence of persistence – the ability to save the state of a Bean and   restore it later.

### 8.2.1 Serialization and Deserialization

Serialization is the ability to save the state of several objects to a stream .The stream is typically assosiated with a file. This allows an application to start execution, read a file, and terminate.

Deserialization is the ability to restore the state of several objects from a stream. This alows and application to start execution read a file, restore a set objects from the data in that file, and continue. If an object contains references to other objects, these are also saved. This is done automatically. The process is recursive, so an attempt to serialize one object can result in the serialization of many other objects. The serialization mechanisms are designed to correctly handle sets of objects that have circular references to each other.

Only the nonstatic and nontransient parts of an object's state are saved by the serialization mechanisms. Static fields of an object are not saved, because they are considered part of the state of the class, not the state of an object. In addition, static fields are some times initialized by static initialization blocks that are executed when the class is loaded. Transient fields of an object are not saved, saved since they contain temporary data not needed to corectly restore that object later. In addition to the object state data, some type information is saved so the object can be reconstructed properly. It is important to understand that when an object is deserialized, none of its constructors is invoked. Instead, memory is allocated and the variables are set directly from the data that is read from the serial stream.

Example :Object Graphs

Consider the collection of objects shown in figure 1.5. There are five objects that hold references to each other as depicted by the arrows. This arrangement is called an object graph. Note that there are cycles in the graph. That is, it is possble to reach some objects by more than one path. For example, D can reach some objects by starting at A and following the references labled "2,"6,"and "8."It can be accessed by starting at A and following the references labeled "2" and"5."we can also tranverse the references labeled "1" and"3" to access D. If you attempt to serialize A, the reference among these objects sause all of them to be saved. Although an object may be encountered several times during this process, it is extremely important that it be saved only once in the serial stream. Otherwise, multiple copies of the same object will be created during deserialization .The java persistence mechanism have been designed to operate in this manner. When a second time, another complete copy of it is not written to the stream. Instead, a handle is written. This is a reference to an object that has already been written to the stream.

Object Graph
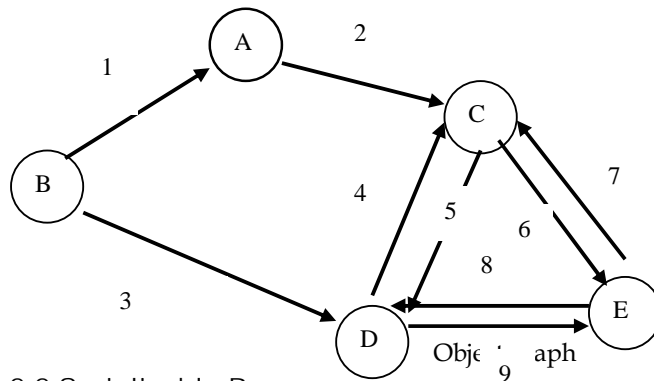
8.2.2 Serializable Bean

This section develops a Bean called graph that shows six nodes positioned at the corners of a hexagon. You can between nodes by moving the mouse to a node, pressing the mouse button, and dragging the mouse to another node and then releasing it.This Bean is serializable because all of its classes either directly or indirectly implements the Serializable interface. It illustrates several issues relating to persistence. The source code for the example is located in three files.

1. Graph.java
2. Node.java
3. Link.java

STEP 1: Creating Graph class

Here we are going to do the following:
Creating a class which extends Canvas and implements MouseListener and MouseMotionListener.The graph class is serializable because it inherits from the Component class, which implements the Serializable interface.

References to the Node and Link objects that make up the graph are maintained in vectors named **nodes** and **links.** Variable node1 and node2 are used only when the user is drawing a link between two nodes. They are transient because we do not want to store this data as part of the serialization process. Variables x and y are used only to track the current position of the mouse aas it being dragged. These are also transient variables because we do not want to store as part of the serialization process.

The constructor begins by setting the size of the Bean to 200 x 200 pixels and initializing the nodes and links vectors. The object registers itself to receive mouse and mouse motion events.

Finally, the **makenodes()** method is called to create the six nodes at the appropriate positions. Impementations are provided for all of the methods in the MouseListener and MouseMotionLIstener interfaces.

The **doMousePressed ()** method checks if the mouse is positioned within a node. If so, the variable node1 is set to reference that node. Otherwise, node1 is set to null.

The **doMousedragged ()** method updates the x and y variables to tack the position of the mouse and invokes repaint(), so that a rubber-band line can be drawn from the center of the first node to the current mouse position.

The **doMouseReleased()** method checks if node1 has been set. If so, it then checks if the mouse is positioned within a node. If so, **makeLink()** is called to create a Link object connecting node1 and node2. In any case, node1 and node2 are set to null, and repaint() is called to update the display.

The **paint()** method draws the nodes, links, and a rubber-band line if needed.

The **makeNodes()** method generates a display that shows the nodes of the graph and adds these to the nodes vector. The makeLink() method returns if a link already exists between node1 and node2. Otherwise, it creates a new Link object and adds it to the links vector. The **linkExists()** methods retirns true if a link already exists between two specified nodes. Otherwise, it returns false.

Listing Graph.java

```
Package graphs;
import java.awt. *;
import java.awt.event. *;


import java.io. *;
import java.util. *;
public class Graph extends canvas
        implements mouseListener,mouseMotionListener
```

```
{
 private final static int NNODES = 6;
 private Vector nodes;
 private Vector links;
 private transient Node node1, node2;
 private transient int x, y;

 public Graph() {
   setSize(200, 200);
   nodes = new Vector();
   inks = new Vector();
   addMouseListener(this);
   addMouseMotionListener(this);
   makeNodes();
       }

 public void mouseClicked(MouseEvent me) {
 }

 public void mouseEntered(MouseEvent me) {
 }

 public void mouseExited(MouseEvent me) {
 }

 public void mousePressed(MouseEvent me) {
   doMousePressed(me);
 }

 public void mouseReleased(MouseEvent me) {
   doMouseReleased(me);
 }

 public void mouseDragged(MouseEvent me) {
   doMouseDragged(me);
 }

 public void mouseMoved(MouseEvent me) {
 }

 public void doMousePressed(MouseEvent me) {
   // check if node1 should be initialized
   X = me.getX();
```

```
      Y = me.getY();
      enumeration e = nodes.elements();
      while (e.hasMoreElements()) {
        node1 = (Node)e.nextElement();
        if (node1.contains (x, y)) {
          return;
        }
      }
      node1 = null;
    }
    public void doMouseDragged(MouseEvent me) {
      X = me.getX();
      Y = me.getY();
      repaint ();
    }


    public void doMouseReleased(MouseEvent me) {
      // make a link between node1 and node2
      X = me.getX();
      Y = me.getY();
      if (node1! = null) {
        enumeration e = nodes.elements();
        while (e.hasMoreElements()) {
          node2 = (Node)e.nextElement();
          if (node 2. Contains (x, y)) {
            makeLink(node1.getId(), node2.getId());
            break;
          }
        }
        node1 = node2 = null;
        repaint();
      }
    }


    public void paint (graphics g) {
      // Draw nodes
      enumeration e = nodes.elements();
      while (e.hasMoreElements()) {
        ((node) e.nextElement()).draw(g);
      }
      // Draw links
      e = links.elements();
      while (e.hasMoreElements()) {
```

```
    ((link)e.nextElement()).draw(g);
  }
  // Draw rubber band line (if any)
  if (node 1!  = null) {
    g.drawLine(node1.getX(), node1.getY(), x, y);
  }
}
private void makenodes() {
  // Initialize nodes variable
  Dimension d = getSize();
  int width = d.width;
  int height = d.height;
  int centerx = width/2;
  int centery = height/2;
  double radius = (width < height)? 0.4 * width:   0.4 * height;
  for (int i = 0; i < NNODES; i++) {
    Double theta = i * 2 * Math.PI/NNODES;
 int x = (int)(centerx + radius * Math.cos(theta));
 int y = (int)(centery - radius * Math.sin(theta));
    nodes.addElement(new Node(x, y));
  }
}

private void makeLink(int id1, int id2) {
// Return if a link already exists between these     nodes
  if(linkExists(id1, id2)) {
    return;
  }
  // otherwise, create a new link
  node n1 = (Node)nodes.elementAt(id1);
  node n2 = (Node)nodes.elementAt(id2);
  links.addElement(new Link(n1, n2));
}

private boolean linkExists(int i, int j) {
  // check if a link exists between nodes i and j
  enumeration e = links.elements();
  while (e.hasMoreElements()) {
    link link = (Link)e.nextElement();
    int id1 = link.getNode1 ().getId();
    int id2 = link.getNode2 ().getId();
  if((id1 == i && id2 == j)||(id1 == j && id2 == i))     {
      return true;
```

```
      }
    }
    return false;
  }
}
```

STEP 2: Creating Node class

Note that this class must implement Serializable.Because the Graph Object holds references to Node objects, any Attempt to serialize the Bean also cause the Node object to be serialized.

The radius of the node is defined by an int constant **NODERADIUS**. Each Node object has an instance variable named id that contains unique value to identify that node. The Node class has a static variable named **count** that is incremented each time a Node Object is Created.This is the mechanism used to assign a unique id value for each node. The x and y variables define the center position of the node.

The Constructor initializes the instance variable. Note that the id variable is set from the current value of the static count variable. Access methods for the x, y and id variables follow the constructor.

The **contains()** method returns true if a point is within node. Otherwise, it returns false. The **draw()** method displays the node.

Listing Node.java

```java
package graphs;
import java.awt.*;
import java.io.*;

public class Node implements Serializable {
private final static int NODERADIUS = 10;
private static int count = 0;
private int x, y, id;

public Node(int x, int y) {
   this.x = x;
    this.y = y;
    id = count++;
```

```
  }

public int getX() {
    return x;
  }

  public int getY () {
    return y;
  }

  public int getId () {
    return id;
  }

  public boolean contains (int x, int y) {
    int deltax = this.x - x;
    int deltay = this.y - y;
    int a = deltax * deltax + deltay * deltay;
    int b = NODERADIUS * NODERADIUS;
    return (a <= b);
  }

  public void draw (graphics g) {
    int w = 2 * NODERADIUS;
    int h = w;
   g.fillOval (x - NODERADIUS, y - NODERADIUS, w, h);
  }
}
```

STEP 3: Creating Link class

Note that this class must be serializable. Because the graph object holds references to Link objects, any attempt to serialize the Bean will also cause Link objects to be serialized.

The node1 and node2 variables hold references to the two nodes connected by this link. The **getNode1()** and **getNode2()** methods provide access to these variables. The **draw()** method draws a line connecting the centers of the two nodes.

```
package graphs;
import java.awt.*;
import java.io.*;
```

```
public class Link implements serializable {
  private Node node1,node2;
  public link(node node1, node node2) {
    this.node1 = node1;
    this.node2 = node2;
  }
  public node getNode1() {
    return node1;
  }
  public node getNode2() {
    return node2;
  }
  public void draw(graphics g) {
    int x1 = node1.getX();
    int y1 = node1.getY();
    int x2 = node2.getX();
    int y2 = node2.getY();
    g.drawLine(x1,y1,x2,y2);
  }
}
```

Note that the Link objects have references to Node objects. Furthermore, a given Node object may be referenced by more than one Link Object.

To run the application, create seperate manifest and jar files for each bean and instantiate them in the beanbox. Note that the Link objects have references to Node objects. Furthermore, a given Node object may be referenced by more than one Link Object.This example has illustrated that object graphs are correctly saved and restored. To confirm this, try saving and restoring the state of this Bean.

From this example we have learnt that, each bean is capable of storing and restoring its state.

8.3 Short Summary

- Serialization is the ability to save the state of several objects to a stream .The stream is typically assosiated with a file.

- Deserialization is the ability to restore the state of several objects from a stream. This alows and application to start execution read a file

- In our example

- The Graph Bean is serializable because all of its classes either directly or indirectly implements the Serializable   interface.

8.4 Brain Storm

1. Define Serialization.
2. Define Deserialization
3. What are the steps involved in Serialization bean?

 හෟ

Lecture 9

Introspection

Objectives

In this lecture you will learn the following

- About Introspection

- About bean descriptor

- About property descriptor

Coverage Plan

Lecture 9

9.1 Snap Shot- Introspection

Introspection is the ability to obtain information about the properties, events, and methods of a Bean. Builder tools use this feature. It provides the data that is needed so developers who are using Beans can configure and connect components. It is also possible to explicitly designate which properties, events and methods are displayed to a user by a builder tool. This is very important and is necessary for building productive quality Beans.

The following sections present an overview of the classes and interfaces that provide functionality.

9.2 Introspector

The Introspector class in the Java.beans package provides static methods that allow the user to obtain information about the properties, events and methods of a Bean. One of the most commonly used methods of Introspector is **getBeanInfo()**.
It has two forms shown below

Static BeanInfo getBeanInfo(Class beanCls)

The above method returns an object that implements the BeanInfo interface. That object describes the properties, events and methods of beanCls and all of its superclass.

9.2.1 BeanInfo

The BeanInfo interface in the java.beans package defines a set of constants and methods that are central to the process of introspection. The int constants defined by BeanInfo are used to identify icons of different sizes that you can provide for a component. The builder tool can use these icons to provide a visual representation of a Bean.

9.2.2 SimpleBeanInfo

The SimpleBeanInfo class in the java.beans package provides a default implementation of the BeanInfo interface. To provide information about a Bean, a developer extends this class and

overrides the implementations of some of its methods. This technique is shown in the code examples.

9.2.3 Feature Descriptor

The FeatureDescriptor class in the java.beans package is the immediate superclass of the BeanDescriptor, EventSetDescriptor, Method Descriptor, and ParameterDescriptor and PropertyDescriptor classes.

9.2.4 BeanDescriptor

The BeanDescriptor class in the Java.beans package associates a customizer with a Bean. A customizer provides a graphical user interface through which a user may modify the properties of a Bean. Its most commonly used constructor is

BeanDescriptor (Class beanCls, Class customizerCls)
The two methods provided by this class are shown here:
Class getBeanClass() and

Class getCustomizerClass().

The getBeanClass() method returns the Class object for a Bean, and the getCustomizerClass() method returns the Class object for a Bean customizer.

9.2.5 EventSetDescriptor

The EventSetDescriptor class in the Java.beans package describes a set of events generated by a Bean. These are one or more events that are processed by an EventListener interface.

The class supports the constructors shown below

```
EventSetDescriptor(Class src,String esName,
Class listener, String listenerMethName)
EventSetDescriptor(Class src,String esName,
Class listener,
String[] ListenerMethNames,
```

```
String addListenerMethName,
 String removeListenerMethName)


EventSetDescriptor(String esName,Class listener,
Method[] listenerMeths,
Method addListenerMeth,
Method removeListenerMeth)
EventSetDescriptor(String esName,Class listener,     MethodDescriptor[]
listener MethDescs,
Method addListenerMeth,
Method removeListenerMeth)
```

The arguments to these constructors have the following meaning:

src is the class of the Bean that generates the event set. *esName* is the name of the event set .

listener is the class of the listener interface, listenerMethName is the name of the listener method.

listenerMethNames are the names of the listener methods.

addListenerMethName is the name of the method used to register a listener.

removeListenerMethName is the name of the method used to unregister a listener.

listenerMeths is an array of method objects describing the listener methods.

listenerMethDescs is an array of MethodDescriptor objects describing the methods in the listener interface.

addListenerMeth is a method object describing the method used to register a listener.

removeListenerMeth is a method object describing the method used to register a listener.

Each of these constructors can generate an **IntrospectionException.**

MethodDescriptor

The MethodDescriptor class in the Java.beans package describes a method of a Bean. The class supports the constructors shown next

MethodDescriptor (Method meth)

Method	Descriptor
Method getAddListenerMethod()	Returns a Method object for the registration method.
MethodDescriptor[] getListenerMethodDescriptors ()	Returns an array of MethodDescriptor objects for the methods in the listener interface.
Method[] getListenerMethods()	Returns an array of Method objects for the methods in the listener interface
Class getListenerType()	Returns a Class object for the listener interface.
Method getRemoveListenerMethod()	Returns a Method object for the unregistration method.
Boolean isInDefaultEventSet()	Returns true if the event set is in the "default set". Otherwise returns false.
Boolean isUnicast()	Returns true if the event set is unicast. Otherwise returns false.
Void setInDefaultEventSet(boolean Flag)	If the flag is true, the event set is part of the default set. Otherwise it is not.
Void setUnicast(boolean flag)	If the flag is true, the event set is unicast. Otherwise it is multicast.

Table 1.1 MethodDescriptor

MethodDescriptor (Method meth, ParameterDescriptor[] pds)

Here, meth is a method object for this method, and pds is an array of parameterDescriptor objects that describe the parameters to this method.

The methods defined by MethodDescriptor are shown here:

Method getMethod()

ParameterDescriptor[] getParameterDescriptors()

The **getMethod()** method returns a method object for the associated method, and getParameterDescriptors() returns an array of ParameterDescriptor objects for the parameters of this method.

ParameterDescriptor

The ParameterDescriptor class in the Java.beans package describes the parameters of a method. The class does not provide any additional fields or methods beyond those of its FeatureDescriptor superclass.

9.2.6 PropertyDescriptor

The PropertyDescriptor class in a Java.beans package describes a property of a Bean. It supports the constructors shown next.

Property Descriptor (String pname, Class cls) Property Descriptor (String pname, Class cls, String getMethName, String setMethName) Property Descriptor (String pname, Method getMeth, Method setMeth)

Here, pname is the name of the property, and cls is the class of the Bean. The names of the access methods for this property are getMethName and setMethName. The argument getMeth, setMeth are method objects for these access methods. These constructors can throw ans IntrospectionException.

Following table illustrates the methods

Method	Description
Class getPropertyEditorClass()	Returns a class object for the associated property editor. If a property editor has not been defined for this property, null is returned. In this case, the PropertyEditorManager is used to obtain a property editor.
Class getPropertyType()	Returns a class object for the property.

Method getReadMethod()	Returns a method object for the reader.
Method getWriteMethod()	Returns a method object for the writer.
Void setPropertyEditorClass(Class pEdCls)	Sets the property editor for this property to pEdCls.

Table 1.2 Property Descriptors

9.2.7 IndexedPropertyDescriptor

The IndexedPropertyDescriptor class in the Java.beans package describes an indexed property of a Bean. It extends the PropertyDescriptor class

9.2.8 An Introspection Example

This section describes how a Bean can explicitly control the properties, events and methods that are presented to a user by a builder tool.

Designating Bean Properties

This example presents an enhanced version of the Spectrum Bean. The Bean here is named Spectrum2. Its source code is identical to that seen previously except for the name change. It is reproduced in the following listing.

```
package spectrum2;
import java.awt.*;
public class Spectrum2 extends Canvas {
  private boolean vertical;

  public Spectrum2() {
    vertical = true;
    setSize (100, 100);
  }

  public boolean getVertical() {
    return vertical;
```

```
  }

  public void setVertical(boolean vertical) {
    this.vertical = vertical;
    repaint();
  }

  public void paint(graphics g) {
    color c;
    foat saturation = 1.0f;
    float brightness = 1.0f;
    dimension d = getSize();
    if (vertical) {
      for (int y = 0; y < d.height; y++) {
        float hue = (float)y/(d.height - 1);
        c = Color.getHSBColor(hue,saturation,
             brightness);
        g.setColor(c);
        g.drawLine (0, y, d.width - 1, y);
      }
    }
    else {
      for (int x = 0; x < d.width; x++) {
        float hue = (float) x/(d.width - 1);
        c = Color.getHSBColor(hue,saturation,
             brightness);
        g.setColor(c);
        G.drawLine(x, 0, x, d.height - 1);
      }
    }
  }
}
```

A Spectrum2BeanInfo class is developed for this Bean. It extends SimpleBeanInfo and overrides the getPropertyDescriptors(), getEventSetDescriptors(), and getMethodDescriptors() methods. This is the manner in which the Bean developer explicitly designates what is presented to a user of a builder tool.

The getpropertyDescriptors() method returns an array of PropertyDescriptor objects. There is only one element in this array. It is the descriptor for the vertical property in the spectrum2 class.

The Bean does not generate or receive any events. Therefore the getEventSetDescriptors() and getMethodDescriptors() methods return an empty array of eventSetDescriptor and methodDescriptor objects respectively.

```
package spectrum2;
import java.beans.*;
import java.awt.event.*;
import java.lang.reflect.*;

public class Spectrum2BeanInfo extends SimpleBeanInfo {

 public propertyDescriptor[] getpropertyDescriptors() {
    try {
      class cls = spectrum2.class;
      propertyDescriptor pd;
      pd = new propertyDescriptor("vertical", cls);
      propertyDescriptor pds[] = {Pd};
      return pds;
    }
    catch(Exception ex) {
    }
    return null;
  }
 public eventSetDescriptor[] getEventSetDescriptors() {
    eventSetDescriptor esds[] = {};
    return esds;
  }
  public methodDescriptor[] getMethodDescriptors() {
    methodDescriptor mds[] = {};
    return mds;
  }
}
```

Figure 1.4 shows how the Spectrum2 appears in the BeanBox. Observe that the properties window contains only one property. Also, if you pull down the Edit menu of BeanBox, you can see that this component generates no events. Finally, if you attempt to map an event generated by other Bean to a method of the Spectrum2 Bean, the BeanBox provides a message box stating that there is no suitable target method.
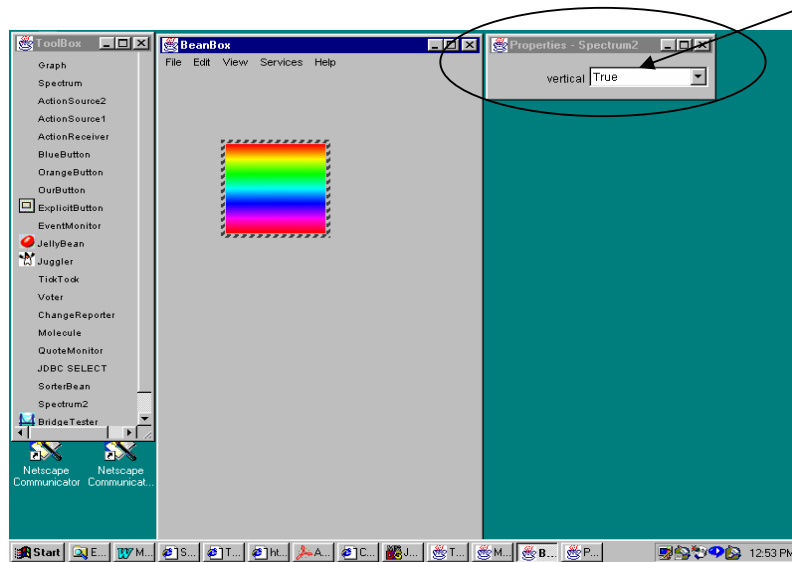
Figure: 1.4 Appearance of Spectrum2 in Bean Box

9.3 Short Summary

- Introspection is the ability to obtain information about the properties, events, and methods of a Bean.

- Introspection is very important and is necessary for building productive quality Beans.

- The Introspector class in the Java.beans package provides static methods that allow the user to obtain information about the properties, events and methods of a Bean.

- The BeanInfo interface in the java.beans package defines a set of constants and methods

- The SimpleBeanInfo class in the java.beans package provides a default implementation

- The BeanDescriptor class in the Java.beans package associates a customizer with a Bean.

- The EventSetDescriptor class in the Java.beans package describes a set of events

9.4 Brain Storm

1. Define Introspection.
2. Define simple beaninfo and simple describtor class
3. Write short notes on Introspection.

ೞఞ

Centre for Information Technology and Engineering, Manonaniam Sundaranar University

Lecture 10

Properties

Objectives

In this lecture you will learn the following

❧ About property

❧ About different types of properties

Coverage Plan

Lecture 10

10.1 Snap Shot - Properties

This section discusses the Various properties available for a Bean.It also discusses how to make bean property of a particular type.

10.2 What is a Property?

Property is a public attribute of a bean that affects appearance or behavior. Typical attributes are like color, font, name, etc. They will usually be persistent and may be presented in a property sheet for editing.

10.3 Different type of properties

10.3.1 Simple Properties

A simple property contains one value that may be either a simple type or an object. Properties are identified/accessed by a pair of get / set methods:

```
public void setN(TYPE value);
public TYPE getN();
```
(Here N is the Name of the property.)

Read-only properties only have a *get* method. Write-only properties only have a *set* method. A property is said to be read-write only if it has both get and set methods.

Example : The following example allows us to construct a circle bean with simple property '*color*'.This is a read-write property as it allows us to read and write the value.

Listing Circle.java

```
import java.awt.*;
import java.beans.*;

public class Circle extends Canvas{
private Color color;
```

```
public Circle(){
      color=Color.green;
      }
public Color getColor(){
      return color;
}
public void setColor(Color val){
      color=val;
}
public void paint(Graphics g){
      setForeground(color);
      Dimension d=getSize();
      int h=d.height;
      int w=d.width;
      g.fillOval(0,0,w-1,h-1);
}
}
```

To run the above example:

- Create .mft file with the following entry

Name: Circle.class
Java-Bean: True

- Create .jar file
- Load the .jar file in BDK

Output   Simple Property

Properties of Canvas(base class)
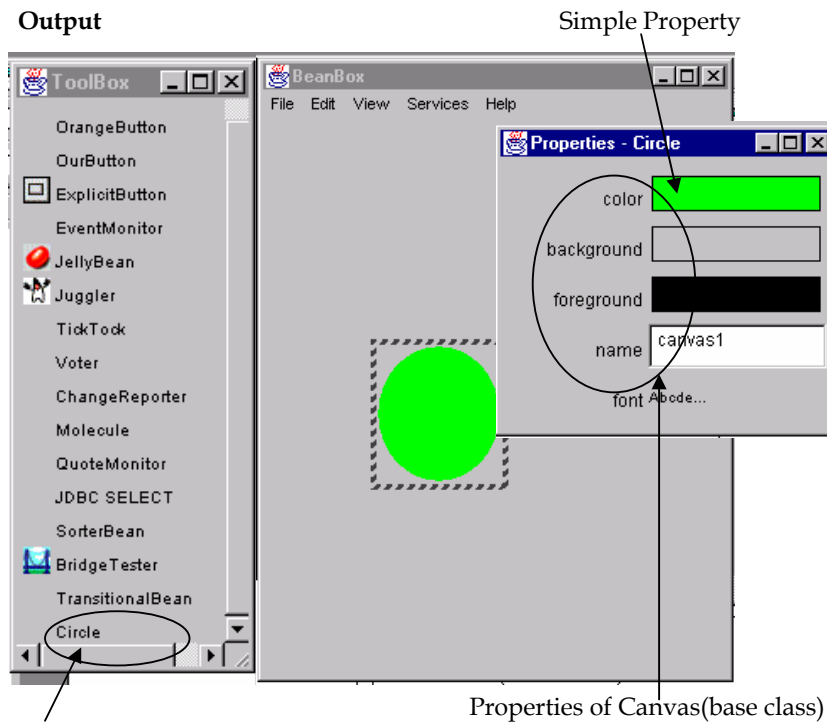
Figure 1.5 Circle Bean

10.3.2 Boolean Properties

A Boolean property contains one value that may be either true or false. The following naming patterns are used for its access methods:

```
public boolean isN()
public boolean getN()
public void setN(boolean value)
```

Here N is the Name of the property .The First and second from can be used to read the property. (If both exists, the first is used) .The third form is used to write the property.A read only property has only the isN() and/or getN() method.A write only property has the setN() method.

10.3.3. Indexed Properties

An Indexed property contains several values that may be either simple types or objects. The following is the naming patterns that are used for its access methods.

```
public T getN(int index)
public T [] getN()
public void setN(int index, T value)
public void setN(T values[])
```

Here, T is the type of property and N is is its name.

- The *first form* reads one value where index identifies which entry is wanted.
- All values may be retrieved with the *second form* .
- *The third form* writes one value. The arguement index identifies which entry to change, and the argument value is the new value for that property.
- All values may be updated with the *fourth form*. The argument values are in an array that contains the new values for the property. A read only property has only the getN() method, and a write only property has only the setN() method.

10.3.3 Bound Properties

A bound property notifies other objects when its value has changed. Each time its value changes, it notifies the interested objects by firing an event that contains the old and new values of the property (along with the property name, of course).

The below mentioned bean uses a bound property. That is whenever you make any change in the property('color') of the bean, it throws an event named PropertyChangeEvent. Now any other bean can listen to this bean by just implementing PropertyChangeListener interface.
Since our Circle bean is the source it gives implementation for two special methods:

addPropertyChangeListener and

removePropertyChangeListener. These methods are involved in adding and removing the listeners respectively.

```
import java.awt.*;
import java.beans.*;
public class Circle extends Canvas{
      private Color color;
      private PropertyChangeSupport pcs;
public Circle(){
```

```
      color=Color.green;
      pcs=new PropertyChangeSupport(this);
}
public Color getColor(){
      return color;
}
public void setColor(Color val){
      Color old=color;
      Color newColor=val;
      // Fire the event whenever setter method is invoked
      pcs.firePropertyChange("color",old,newColor);
      color=val;
}
public void paint(Graphics g){
      setForeground(color);
      Dimension d=getSize();
      int h=d.height;
      int w=d.width;
      g.fillOval(0,0,w-1,h-1);
}
public void
addPropertyChangeListener(PropertyChangeListener pcl){
       pcs.addPropertyChangeListener(pcl);
}
public void
removePropertyChangeListener(PropertyChangeListener pcl){
      pcs.removePropertyChangeListener(pcl);
}
}
```

Listing Circle.java

Now it is time to write a listener bean. This bean implements PropertyChangeListener and thus capable of listening to PropertyChange event. The Square bean actually changes its property according to the change in property of the Circle bean.

```
import java.awt.*;
import java.beans.*;
public class Square extends Canvas implements
PropertyChangeListener{
private Color color;
```

```
public Sequence(){
       color=Color.blue;
}
public Color getColor(){
       return color;
}
public void setColor(Color color) {
        setForeground(color);
        repaint();
}
public void propertyChange(PropertyChangeEvent e){
// The event describes the property change of the Circle bean So get the
// new value and assign it to the Square bean.
       color=(Color)e.getNewValue();
       repaint();
}
public void paint(Graphics g){
       Dimension d=getSize();
       setForeground(color);
       int h=d.height;
       int w=d.width;
       g.fillRect(0,0,h-1,w-1);
}
}
```

10.4 Short summary

- Read-only properties only have a *get* method.
- Write-only properties only have a *set* method.
- A property is said to be read-write only if it has both get and set methods.
- A bound property notifies other objects when its value has changed.
- A Boolean property contains one value that may be either true or false

10.5 Brain Storm

1. Define property
2. What are the property types?
3. Explain simple property.
4. Write sort note on boolean and bound property.

ೞೞ

Lecture 11

Constraints & Customizations

Objectives

In this lecture you will learn the following

- About Customization

- Need of Customization

- Customization levels

Coverage Plan

Lecture 11

11.1 Snap Shot - Constrained Properties

Constrained properties are properties that permit other objects to validate changes in value.

Creating a Constrained property.

Constrained properties have the following pattern:

```
public TYPE getXXX()
public void setXXX(TYPE value)
     Throws PropertyVetoException
```

Where PropertyVetoException is thrown if the validation fails.
You need to implement a pair of methods to maintain list of vetoable change listeners:

```
public void addVetoableChangeListener(VetoableChangeListener x)
public void removeVetoableChangeListener(VetoableChangeListener x)
```

The utility class VetoableChangeSupport is available to do most of the work for you, managing the list and providing notification support.

The registration support would look like:

```
private VetoableChangeSupport vetoes = new
VetoableChangeSupport(this);

public void addVetoableChangeListener(
        VetoableChangeListener v) {

   vetoes.addVetoableChangeListener(v);
  }

  public void removeVetoableChangeListener(
        VetoableChangeListener v) {
   vetoes.removeVetoableChangeListener(v);
  }
```

The constrained property would then look something like the following:

```
public void setMood(int mood)
    throws PropertyVetoException {
  vetoes.fireVetoableChange("mood",
    new Integer(this.mood), new Integer(mood));
  int old = this.mood;
  this.mood = mood;
  repaint();
}
```

And the object interested in validating the change would need:

```
public void vetoableChange(
        PropertyChangeEvent e)
  throws PropertyVetoException {
 ...
}
```

Frequently, a constrained property is also bound. This would result in the following property setter method:

```
public void setMood(int mood)
    throws PropertyVetoException {
  vetoes.fireVetoableChange("mood",
    new Integer(this.mood), new Integer(mood));
  int old = this.mood;
  this.mood = mood;
  repaint();
  changes.firePropertyChange("mood",
        new Integer(old), new Integer(mood));
}
```

11.2 Customization

What is customization?

customization means configuring the *internal state* of a bean so that it appears and behaves properly in the situation in which it is being used. The individual elements of this internal state (color, size, password string, and so on) are called *properties* in the JavaBeans Spec.

11.3 Need for Customization

A software component can be used in a wider range of applications when the application developer has control over its appearance and behavior. For example, a PushButton class wouldn't be very useful if its text label were always the word "Button," and its associated action were always, say, to reboot your machine (although with some operating systems, this might be one of your most useful tools). Even a component as simple as a lowly PushButton may have many attributes that a developer might want to control, including:

- whether or not it is enabled
- action
- background color
- text color
- size
- position
- shape
- label text (or maybe an icon instead of a label)
- the sound file to play when the button is pressed

Complex components have even more involved customization requirements. Customizing a remote database connection might entail selecting from a list of available servers (information available only at run time), choosing a protocol (ditto), specifying user name and password, and setting up access through a firewall.

11.4 PropertyEditor and Customization.

An IDE that complies with the JavaBeans Spec knows how to analyze a bean to discover its properties. It also knows how to create a visual representation for each property type, called a *property editor*, to allow the application developer to modify the properties at design time. When the developer drops a bean into an application, the IDE draws the bean on the panel. It then presents a *property sheet*, which is simply a list of all of the bean's properties, with associated editors for each property. The IDE will call all of the *getter* methods so that the property sheet contains the bean's current property values. If the developer changes a property in the property sheet, the IDE calls the corresponding *setter* method to update the associated property of the selected bean.

For beans of low to moderate complexity, configuring a bean at design time by calling its setter methods is often enough. To support the higher degree of customization control required by more complex beans, though, the Spec defines a *customizer* class that developers may extend and include with a bean. This leaves the door wide open for customization of any sort: sophisticated color pickers, graphical network configuration wizards, even a VRML interface if you're up to writing one! Actually, the property sheet described above can be considered a customizer that is automatically generated by an IDE. You'll see this pattern often in the JavaBeans framework: Simple things (in this case, simple property types for which property editors already exist) usually entail little or no work, but access is provided "under the hood" for more complicated situations.

11.5 Levels of customization

As beans become more complex, the number of properties (and, therefore, the number of getter and setter methods) begins to explode. More and more properties appear on the dialogs, and soon the dialogs we use to customize beans are oversized, awkward to use, and confusing.

It may not be immediately clear to the user of the bean what all the different properties mean or how they relate to one another. Some properties, particularly *indexed properties*, may have no reasonable default screen representation. Others may simply be more "friendly" if edited graphically.

There are several levels of customization control, ranging from conforming to design patterns and letting the IDE create its own dialog, to writing a customization class that provides a complete GUI for configuring the bean.

Customization level 1: Using design patterns

Accessor (or getter and setter) methods change the internal state of a bean. The JavaBeans Spec specifies that, for each property (let's say it's called Property and is of type SomeType), there should be getter and setter methods that look like this:

- SomeType get*Property*() (getter)
- void set*Property*(SomeType value) (setter)

If a JavaBean conforms to the JavaBeans Spec, an IDE can analyze the bean, find methods whose names conform to the above pattern, and figure out what the properties are and how to set them. It can even whip up a customization dialog box on the fly, since it knows the property names, their types, and how to set and get them. This is the "standard" way that IDEs and other bean containers figure out how to present a bean to an application developer for customization.

Example: Let us look at the Circle bean we have created previously. This bean actually supports Customization level 1. Note that the Circle bean has both getColor and setColor methods.

Unwanted Properties
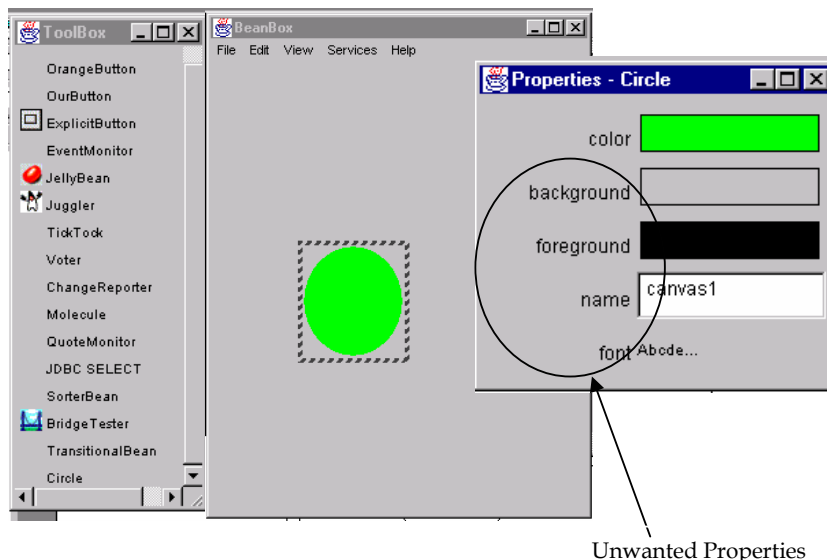
This is a pretty simple bean, but look at that list of properties - background, foreground, and font. Now, imagine trying to configure a spreadsheet or word processor bean with this method: There could be literally hundreds of properties, all arranged in one long column.

It would be best to hide properties that are of no interest to the user or that would actually cause problems for someone if fiddled with.

Customization level 2: Providing a BeanInfo object

The second level of customization, the interface java.beans.BeanInfo, addresses all of these problems. The BeanInfo interface lets the bean describe itself to anyone. In addition to describing properties, the BeanInfo interface can provide the IDE with a display name, an icon, lists of properties, methods (including constructors), event sets, and other information. There are some properties that would simply be better if displayed in a way that's not setting an integer, color, or text field. One common example is an enumerated value: Like choosing between several different colour schemes ("Spring Morning", "Industrial Waste," and so forth) from a drop-down list. Design patterns and BeanInfo simply don't handle these cases directly, but the interface java.beans.PropertyEditor provides the programmer, with a way to present the configuration of the property as a GUI object; in other words, you can write a *property editor*.

Customization level 3: Providing custom property editors

The third level of customization, java.beans.PropertyEditor and java.beans.PropertyEditorSupport, provides a way for you to write your own property editor. A property editor is a component that edits a single property of a certain type. When a Color property is clicked to configure the color, the standard property sheet in the BeanBox pops up the standard property editor for properties of the type Color:

The PropertyEditor interface (and PropertyEditorSupport convenience class) lets you specify a property editor of your own devising if the one the IDE provides doesn't suit your needs. The PropertyEditor interface also helps you to implement enumerated types by its support of "tags," the text values in the enumerated type.

Customization level 4: Providing a Customizer

The fourth and final layer of customization control is the interface java.beans.Customizer, with which you completely replace the IDE's standard property sheet for the bean. You write a custom companion class that is delivered with the bean, and that class is used to configure the bean in place of the standard property sheet.

If designed properly, customization dialogs for beans could be used as subcomponents in user interfaces in a target application. Imagine for a moment you had a PrintBean component that handled scheduling print jobs. Which interface would you rather use?

11.6 Short Summary

- As beans become more complex, the number of properties (and, therefore, the number of getter and setter methods) begins to explode.

- customization means configuring the *internal state* of a bean

- The PropertyEditor interface also helps you to implement enumerated types by its support of "tags," the text values in the enumerated type.
- A property editor is a component that edits a single property of a certain type.

- The interface java.beans.Customizer completely replace the IDE's standard property sheet for the bean.

11.7 Brain Storm

1. What is Persistence? How it is achieved in Java Beans?
2. What is a Property Editor?
3. What do you mean by Customizers?
4. How is a bean different from an Applet and an Object?
5. hy Introspection is needed for Beans?

৪০৫৪

`

Lecture 12

Discussion

Lecture 13

Enterprise Javabeans

Objectives

In this lecture you will learn the following:

- 🔖 Introduction to Middle ware architecture

- 🔖 Application Server

- 🔖 Middle ware architecture

- 🔖 Application server

13.1 Snap shot

This chapter provides an overview of the Middle ware Architecture, features of Enterprise JavaBeans, architecture of EJB, the component model of EJB, deploying a EJB and the roles and responsibilities involved in deploying Enterprise JavaBean. This chapter also illustrates how to create a simple Enterprise JavaBean and also the steps involved in creating a bean.

13.2 EJB – Overview

The Enterprise JavaBeans specification--created by the JavaSoft division of Sun Microsystems--defines an application programming interface (API) that promises to simplify the development, deployment, and management of multi-tier, cross-platform, distributed object applications. Using the Enterprise JavaBeans API, developers can focus on writing business logic for middle-tier servers while spending less time coding and testing on the infrastructure aspects of a distributed application. Because each Enterprise JavaBeans component encapsulates an essential business function, developers do not need to know how to write specialized system-level programs that control features such as security and the ability to handle multiple transactions--often tedious and complex tasks.

However, EJB is simply a model. Critical implementation decisions are left to the vendors who are providing an EJB solution. IT departments need an EJB implementation that meets the needs of enterprise-class applications. This solution must:

- Use industry-standard protocols
- Integrate with familiar IDEs
- Support transactions in a distributed environment
- Enable connectivity from a variety of clients
- Provide security, fault tolerance, and scalability

A product that integrates these qualities is best described as an *application server*, which is dealt I detail Later.

13.2.1 The Client/server architecture

The client/server architecture is one of the most common solutions to the conundrum of how to handle the need for both centralized data control and widespread data accessibility. In client/server systems, information is kept relatively centralized (or is partitioned and/or replicated among distributed servers), which facilitates control and consistency of data, while still providing access to the data users need.

Client-server systems are now commonly composed of various numbers of *tiers*. The standard old mainframe or timesharing system, where the user interface runs on the same computer as the database and business applications, is known as *single tier*. Such systems are relatively

easy to manage, and data consistency is simple because data is stored in only one place. Unfortunately, single-tier systems have limited scalability and are prone to availability hazards (if one computer's down, your whole business goes down), particularly if communication is involved.

The first client/server systems were *two-tier,* wherein the user interface ran on the client, and the database lived on the server. Such systems are still common. One garden-variety type of two-tier server performs most of the business logic on the client, updating shared data by sending streams of SQL to the server. This is a flexible solution, since the client/server conversation occurs at the level of the server's database language. In such a system, a properly designed client can be modified to reflect new business rules and conditions without modifying the server, as long as the server has access to the database schema (tables, views, and so forth) needed to perform the transactions. The server in such a two-tier system is called a *database server,* as shown below.
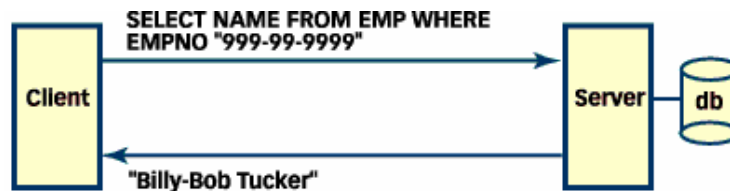

Figure 2.1 A database server

Database servers have some liabilities, though. Often the SQL for a particular business function (for example, adding an item to an order) is identical, with the exception of the data being updated or inserted, from call to call. A database server ends up parsing and reparsing nearly identical SQL for each business function. For example, all SQL statements for adding an item to an order are likely to be very similar, as are the SQL statements for finding a customer in the database. The time this parsing takes would be better spent actually processing data. (There are remedies to this problem, including SQL parse caches and stored procedures.) Another problem that arises is versioning the clients and the database at the same time: all machines must shut down for upgrades, and clients or servers that fall behind in their software version typically aren't usable until they're upgraded.

How Did We Get Here?

As distributed object computing has taken hold, new opportunities have been presented to Information Technology (IT). Distributed object architectures, like CORBA, have enabled IT

to produce applications in record time for increasingly heterogeneous platforms, incorporating diverse sources of data. By partitioning applications into components, and shifting business logic onto middle-tier servers, IT has vastly reduced turnaround time for development of applications by reusing the logic in these middle-tier server objects. Yet, relatively few organizations have converted to middle-tier servers because the complexity of coding multi-tier applications requires staff that understand specialized system-level programming.

However, the phenomenal growth of Web-based computing is driving IT to use the multi-tier approach. Web-based business applications require a thin-client architecture to support browser-based clients. These clients need to interact with intranet-based resources, but often have limited system resources that make it difficult to download applets. To alleviate the burden from these clients, IT departments want to build portable server-side solutions that bridge heterogeneous platforms and integrate with legacy systems. As Java and CORBA become mainstream, IT departments are looking to incorporate these technologies into their solutions.
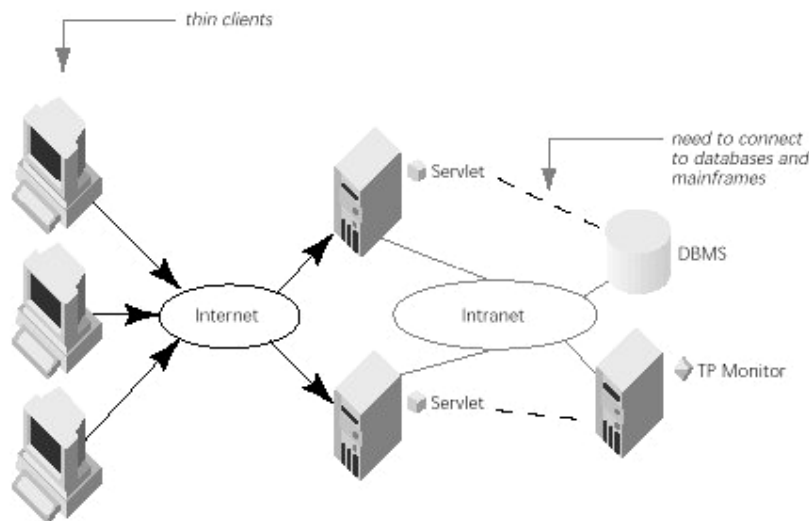
Figure 2.2: Web-based applications require thin-clients that use robust servlets to access back-end databases and legacy systems.

To meet these growing demands, IT departments need a "write once, run everywhere" development model that is based on the industry-standard Java and CORBA technologies. They need a solution that simplifies the development, deployment, and management of these multi-tier applications so that more application developers can build sophisticated distributed applications--without needing to handle complexities such as transactions and security.

13.2.2 Component Transaction Monitors

The CTM industry grew out of both the ORB and the transaction processing monitor (TP monitor) industries. The CTM is really a hybrid of these two technologies that provides a powerful, robust distributed object platform. The following sections explain both TP monitors and ORBs.

13.2.3 TP Monitors

Transactions processing monitors have become powerful, high-speed server platforms for mission-critical applications. TP monitors are operating systems for business systems written in languages like COBOL. They are termed as operating system as they manage the entire environment that a business system runs in, including memory, database access, and transactions.

The business logic in TP monitors is made up of procedural applications that are often accessed through network messaging or remote procedure calls (RPC), which are ancestors of RMI. Messaging allows a client to send a message directly to a TP monitor requesting that some application be run with certain parameters. Messaging can be synchronous or asynchronous, meaning that the sender may or may not be required to wait for a response. RPC is a distributed mechanism that allows clients to invoke procedures on applications in a TP monitor as if the procedure was executed locally.

The primary difference RPC and RMI is that RPC is procedural and RMI is object oriented. It means that with RMI, methods are invoked on a specific object identity whereas in RPC a client calls procedures on a specific type of application. There is not concept of object identify in RPC as in RMI.

TP monitors work with procedural code that can perform complex tasks but has no sense of identity. Accessing a TP monitor through RPC is like always executing static methods; there's no such thing as a unique object.

13.2.4 Object Request Brokers

Distributed objects allow unique objects that have state and identity to be distributed across great distances so that they can be accessed by other systems. Distributed object technologies like CORBA and Java RMI grew out of RPC with one significant difference: distributed object methods are invoked on an object instance, not an application procedure. Distributed objects are usually deployed on some kind of ORB, which is responsible for helping client applications find distributed objects easily.

ORBs, however, do not define an "operating system" for distributed objects as RPC does. They are simply communications backbones that are use to access and interact with unique remote objects. While developing a distributed object application the developer is responsible for concurrency, transactions, resource management and fault tolerance. Though system-level functionality is handled automatically, the lack of implicit system-level infrastructure places an enormous burden on the application developer.

13.3 Middle - Ware Architecture

13.3.1 Application Server

An application server is a middle-tier application that combines three components: pieces for communicating with back-end systems (e.g. business applications or databases); pieces for communicating with front-end clients (often, but not necessarily Web clients); and a framework upon which business logic can be hung. Because many databases cannot interpret commands written in HTML, the application server works as a translator, allowing, for example, a customer with a browser to search an online retailer's database for pricing information.

The result is a system that is modular, highly scalable, robust, and dynamic enough to meet the needs of today's businesses. Application servers are seen as filling a large and growing market; more than 25 companies now offer such products.

13.3.2 Example - Application Servers

There are many different types of application servers in common use today, and each provides a container for some type of server-based request. For example:

- A TP monitor contains transactions and manages shared resources on behalf of a transaction. Multiple transactions can work together and rely on the TP monitor to coordinate the extended transaction.

- A database management system (DBMS) contains database requests. Multiple database clients can submit requests to the database concurrently and rely on the DBMS to coordinate locks and transactions.

- A Web server contains Web page requests. Multiple Web clients can submit concurrent page requests to the Web server. The Web server serves up HTML pages or invokes server extensions or servlets in response to requests.
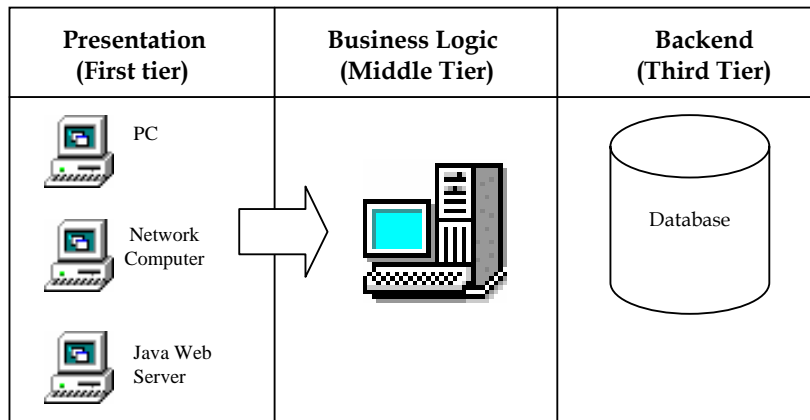
13.3.3 The Transactional and n-tier View

In common usage, application server has come to mean a server that operates in a tier between a client and database server. Application servers that integrate with TP` software are often called transactional application servers. Because these servers are programmed to understand transaction semantics, they are powerful tools for distributed OLTP. Transactional servers are not, of course, the only types of application server. When partitioning applications, middle-tier servers are a prime location for putting non-presentation logic and business rules. For TP, n-tier architecture often combines application servers with transaction managers, database servers, Web servers, and, sometimes, messaging middleware. To distribute an application's workload across several servers, you can put specialized logic and rules in different application servers. nasdaq.com, for example, uses one type of server for disclosure documents and another type as a quote server.

13.3.4 The Middleware and 3-tier View

The World Wide Web may not only be the biggest technical revolution of the 20th century, it may also inadvertently spawn a monumental change in the structure of enterprise computing -- namely, the shift to a three-tier Web-based computing paradigm. That shift is seeing client/server-based applications being usurped in favor of browser-based and thin-client

applications so businesses can take advantage of the considerable capabilities of Internet technology. The main strengths and benefits of the three-tier model include platform independence; universal access to data; ease of application development and deployment; and, consequently, cost-effectiveness. The *application server* is the centerpiece of the new model. Although there are many definitions circulating today, an application server can basically be described as middleware located between clients and data sources and/or legacy systems that manages and processes applications in a three-tier fashion. The application server's main function is to ensure that developed and deployed applications are scalable, reliable, and accessible. And by simply integrating with an HTTP server, the application server can also be used to manage Web-enabled applications. One of the main attractions of application servers is that unlike so many other new technologies, they do not require businesses to replace their existing systems; instead, they help to tie together all of a company's diverse computer systems, both old and new, at one central point. In this way, application servers make it easier for IT departments to share data from many disparate sources with both their internal and external customers.

Presentation (First tier)	Business Logic (Middle Tier)	Backend (Third Tier)
PC Network Computer Java Web Server		Database

Figure 2.3 Three-tier architecture

13.3.5 Why Application Servers?

Application servers offer server-side support for developing and deploying business logic -- business logic that may be located on the server or, more often, partitioned across client and server. This is nothing new: Enterprises rely daily on server-side business processing, ranging from mainframe transaction systems to client/server DBMS stored procedures. Running business processes on the server provides the following:

- **Re-use.** A variety of client applications (HTML-only, Java applets, COM+ components, etc.) can share the same business logic.

- **Intellectual property protection.** Sensitive business logic often includes or manages trade secrets that could potentially be reversed engineered.

- **Security of business logic.** By leaving the logic on the server, user access can be controlled dynamically, revoked at any time.

- **Security of network communications.** Application servers allow use of internet-standard secure protocols like SSL or HTTPS in place of less secure proprietary DBMS protocols.

- **Manageability.** Server-side applications are easier to monitor, control, and update.

- **Performance.** Database intensive business logic will often perform much better when located near the database, saving network traffic and access latency.

- **Download time.** I'Net (Intranet + extranet + Internet) clients most often require access to many different business processes that could require substantial network bandwidth and client memory to download all logic to the client.

- **Compute load.** Running compute-intensive applications on servers saves client cycles.

With the explosive growth of the I'Net, there is unfulfilled demand for application server technology to accompany the now pervasive web infrastructure -- application server technology that goes well beyond CGI and its successors.

13.3.6 What Application Servers should provide?

All of a sudden it seems everyone's got one to sell or to share, promising magical scalability, but since it's been so hyped, a solid definition is hard to find. Essentially, an Application server **is a general-purpose software serve**

13.4 Short summary

- The first client/server systems were *two-tier*

- An application server is a middle-tier application that combines three components…

- A Web server contains Web page requests….

- RPC is procedural and RMI is object oriented…

13.5 Brain Storm

1. Explain the need of application server.
2. Explain middle ware Architecture.
3. Write short notes on EJB.
4. What is the need of Application.
5. What are the goals of EJB?
6. What do you mean by Middleware?
7. What are Component Transaction Monitors?
8. What do mean by "making thin Client "? What are the advantages of making the client thin?

ഔ…ൠ

Lecture 14

EJB - Architecture

Objectives

In this lecture you are going to learn the following

- Need of EJB

- EJB Architecture

- EJB Features

Coverage Plan

## Lecture 14

14.1 Snap Shot - Enterprise Javabeans

The Enterprise JavaBeans (EJB) technology defines a model for the development and deployment of reusable Java server components. Components are pre-developed pieces of application code that can be assembled into working application systems. Java technology currently has a component model called JavaBeans , which supports reusable development components. The EJB architecture logically extends the JavaBeans component model to support server components.

14.2 Why Do We Need EJB?

Ever since the Web began to make the Internet popular and useful for almost everyone (not just the government and students), new I'Net (Internet, Intranet and Extranet) technologies have emerged from all directions and organizations. It is close to impossible to keep up with these technologies at the speed they are being introduced. So, why do we need EJB to add to this growing list of technologies? Well, the answer is for various reasons but before we look at the reasons for EJB, let's look at a couple of reasons why we need Middleware or more specifically, application servers.

Glue Between Front-end and Backend

On the frontend, there are many different types of clients (desktop/browser, PDAs, web phones, NCs/thin clients, different platforms). On the backend, there are many types of platforms (Unix, Windows, Mac) and data repositories (RDBMS, ODBMS, Mainframe); Java Middleware connects the two (see Figure 2.4). In other words, business logic can be implemented as reusable components in the middle tier thereby providing a variety of
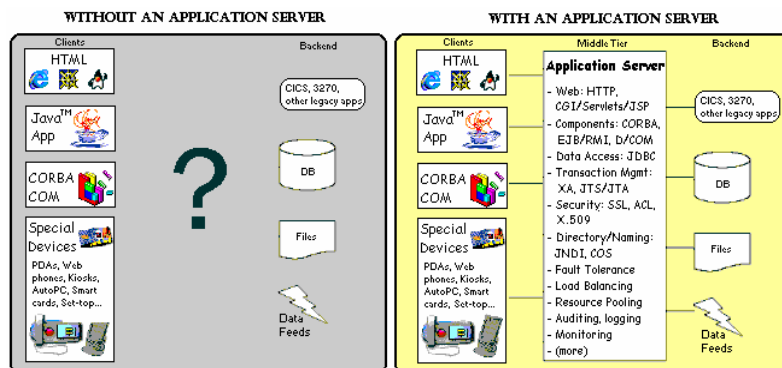
client's access to all types of data on the backend.

Figure 2.4: What an Application Server Provides

Scalability and Standards Based Technologies

The original HTML/CGI solution is not scalable and APIs such as NSAPI, ISAPI are proprietary. Most App Servers provide standards based technologies that scale well. Not to mention, these products also typically provide features such as resource pooling, fault-tolerance, security, management, and other bells and whistles built-in.

One-Stop Shop

Many application servers are a "one stop shop," that is, they support all the necessary protocols to provide a complete application development solution. Take for example, BEA's WebLogic application server which provides support for HTTP, RMI, EJB, Servlets, JNDI, JDBC, and other protocols; Oracle's Application Server provides support HTTP, CORBA/IIOP and more. Hence, you will most likely only need and run one server instead of running multiple ones such as a web server, a CORBA ORB, a RMI server, and more. Now that we have looked why application servers are needed for I'Net development, let us look at why there is a need for EJB.

Productivity

EJB increases productivity for developers because they do not worry about low level system programming (such as connection pooling, security, transaction management, state management, persistence, number of clients, multi-threading) – they simply concentrate on writing the business logic and develop the bean almost as if it will be used by a single client.

Open Server-Side Component Architecture

In the past two or more decades, most server-based products have used the vendor's proprietary APIs and hence not many off-the-shelf, plug-and-play components have been available for these products except from the vendors themselves. EJB changes this by providing portability across platforms and vendors. Because EJB sets a clear path for application vendors, all vendors provide the same minimal functionality in their server products and it opens the floodgates for component builders to build off-the-shelf server-side components, not just client-side GUI components. For example, an off-the-shelf enterprise bean could handle functions such as credit card validation.

Object-Oriented Programming on the Server

Thanks to Java's object-oriented roots and the EJB component model, organizations can achieve strong reusability benefits. For one reason, the logic and data are together in objects. Additionally, EJB containers can translate relational data into objects automatically. This eliminates the distinction between accessing data from database versus any other object.

Java in the Middle Tier

EJB brings all other Java features to the middle tier (such as security, directory services, and serialization). In general, EJB drives Sun's "Layer and Leverage" philosophy forward, which basically calls for portability and leveraging existing enterprise investments.

Support for Other Languages and CORBA

EJB provides support for other languages and CORBA because the middleware vendor handles the communication protocol issues not the bean developer. For example, the default wire protocol for EJB is RMI, however the EJB 1.0 specifications also provide mapping to CORBA. Furthermore, a vendor can use any wire protocol to support many types of clients (e.g. COM/DCOM).

14.3 What Exactly Is EJB?

EJB is part of Sun's Java Platform for the Enterprise (JPE) initiative. Similar to how JavaBeans brought a standard way of developing and using Java components on the client side, EJB is the Java component architecture for the server side. EJB enables software developers to build server-side, reusable business objects. However, EJB takes the notion of reusable objects one step further by providing for attribute based programming to dynamically define lifecycle, transaction, security, and persistence behavior in EJB applications. For example, using attributed based techniques the same EJB component (in it's binary form, that is, class/JAR files) can exhibit different transactional behavior in different applications. Additionally, the method of persisting EJB can be altered during deployment without ever having to re-code the bean. For example, an Entity bean could use a relational database for its persistence or CICS, the bean developer never needs to know.

To provide a more concrete picture of what EJB is, let's look at the following topics taken from the EJB 1.0 specifications:

- Goals
- Architecture
- Features
- Roles and Responsibilities

Goals

The EJB 1.0 specifications define certain goals for Java middleware vendors who will implement this standard. The following are a few of these goals:

- Standard distributed component architecture for Java.
- Portability across platforms and vendors.
- Increased productivity via simplicity (developer does not worry about state management, multi-threading, network connections and protocols, resource pooling, etc.).
- Compatibility with other programming languages and CORBA.
- Address development, deployment and management.
- Compatibility with existing enterprise infrastructure/platform investments.
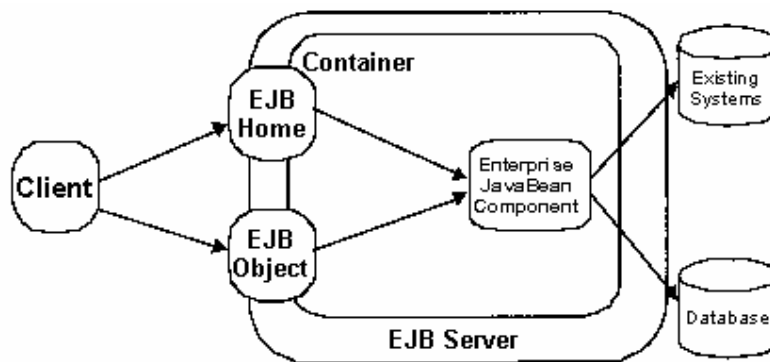
14.4 EJB Architecture

Figure 2.5 EJB Architecture

The figure above illustrates the architecture of EJB technology. The EJB specification allows for any kind of client. This is because the specification does not mandate any remote object "wire" protocol. This means that a server can support multiple protocols.

The EJB server is a collection of services for supporting an EJB installation. These services include management of distributed transactions; management of distributed objects and distributed invocations on these objects, and low-level system services. In short, an EJB server manages the resources needed to support EJB components. An EJB server provider can provide an implementation of a container, (described below) and it can provide an API for third party vendors to plug-in additional EJB containers. The EJB specification allows developers a great deal of freedom in the design and implementation of servers.

An EJB container is just that: a home for EJB components. A container is where a Bean lives, just as a record "lives" in a database. It provides a scalable, secure, transactional environment in which Beans can operate. It is the container that handles the object life cycle, including creating and destroying an object. The container, among other things, also handles the state management of Beans.

A container is transparent to the client. There is no client API to it. When a Bean is installed in a container, the container provides two implementations: an implementation of the Bean's EJBHome interface, discussed below; and the Bean's remote interface. The container is also responsible for making the Bean's EJBHome interface available in JNDI, the Java Naming and Directory Interface.

To construct a Bean, you must first implement the business methods. For example, if you are writing a checking account Bean, you might implement a "debit" method as part of its interface. You must also implement one of two types of EJB interfaces, SessionBean or EntityBean. These interfaces include methods related to working set management, for example, and are not exposed to a client.

To this end, when a Bean is installed on a server, the remote interface, usually called a skeleton in CORBA, is automatically generated. The implementation of the remote interface is called the EJBObject and is an object that exposes only the remote interface specified by the programmer. The enterprise Bean class does not implement the remote interface, though it does contain methods with the same signatures. The EJBObject acts like a proxy, intercepting

the remote object invocations and calling the appropriate methods on the enterprise Bean instance. An EJB container implements the EJBHome interface of each enterprise Bean installed in the container. It allows for the creation of a Bean, deletion of a Bean and querying information or "metadata" about a Bean. The container makes the EJBHome interfaces available to the client through JNDI. For entity Beans, the EJBHome interface also contains one or more "finder" methods that allow a client to look up a Bean by a primary key.

14.5 EJB Features

To fulfill the above goals, EJB provides several features that make this distributed component architecture so appealing. Table 2.1 provides a snapshot of these features (more details on each feature are provided further in the article).

Feature	Supported Through
Component Model	Session Beans Entity Beans
Object Persistence	Entity Beans (EJB Containers)
Transaction Management	JTS/JTA javax.jts.UserTransaction Can be vendor proprietary
Exception Handling	Client and server side
Security	java.security Security related methods in javax.ejb.EJBContext Deployment descriptor properties (Java security ACLs, *RunAs* properties)
Naming and Directory Service	Java Naming and Directory Interface (JNDI)
Wire Protocol	RMI/JRMP IIOP (via CORBA mapping) (Any other wire protocol)
Support for CORBA	CORBA mapping (ejb.idl) CORBA services
Attribute Based Programming	Deployment Descriptor File
Deployment	EJB JAR file

Table 2.1: EJB Feature List

Component Model

EJB provides the three bean types, **Stateless Session Beans, Stateful Session Beans** and **Entity Beans**. Each of these is briefly described here.

Stateless Session Bean is intended to be simple and "light weight" components. Any state, if required, is maintained by the client, thereby making the server highly scalable. Since no state is maintained in this bean type, stateless session beans are not tied to any specific client, hence any available instance of a **Stateless Session Bean** can be used to service a client.

Stateful Session Bean provides easy and transparent state management on the server side. Because state is maintained in this bean type, the app server manages client/bean pairs. In other words, each instance of a given bean is created on behalf of a client and is intended to be a private resource to that client (although it could be shared across clients using the bean instances handle). In essence, a stateful session bean is a logical extension of the client except some of the client's load is distributed between itself and the bean on the server. Any conversational state related data in the object's variables does not survive a server shutdown or crash; although a vendor could provide an enhanced implementation that makes shutdowns and crashes transparent to the client by maintaining the bean's state. **Stateful session beans** can access persistent resources (e.g. databases, files) on behalf of the client, but unlike entity beans, they do not actually represent the data. For example, a session bean could access data via JDBC or an **entity bean**.

Entity Bean is persistent objects and represents an object view of data stored in permanent storage. Perhaps, the best way of to understand what an entity bean is to think of it as a row in a relational database. Along those lines, an entity bean can be created, found and removed -- using the createXXX, findXXX and remove methods -- similar to how a database row can be INSERTed, SELECTed and DELETEd in a SQL database. An entity bean lives in an EJB "Container", similar to how a record lives in a database. Herein lies the beauty of entity beans because a vendor can provide an EJB containers for various data sources (e.g. Oracle, CICS) and the bean developer never needs to know. Unlike stateful session beans, multiple clients can access entity beans concurrently -- the EJB container manages the concurrency. Since entity beans provide an object view of the data, the actual data they map to in the permanent store can either be created via the entity bean's create method or may exist before hand in the permanent store (for example, using a SQL database's INSERT command). On the flip side,

entity beans can be removed either using the entity bean's remove method or be deleted directly from the permanent store. For finding existing beans, the findXXX methods can be used to return a single object or collection of objects.

Deciding between what type of bean to use is more of a design issue than a development one. The following are a couple of general rules of thumb:

- Use **Entity** beans versus **Session** beans if you do not want to embed database calls in your bean classes and also want to take complete advantage of the EJB features such as automatic transaction management, resource pooling, container managed persistence. Also, think of Session Beans as a logical extension to the client and are short lived – the object disappears either because the client shutdown or the server does. For entity beans, think of them as persistent objects that can be shared by multiple clients and long lived, possibly for years (similar to data in a database).

- If you choose to use session beans, picking between **STATELESS** and **STATEFUL** beans is matter of deciding how many loads to put on the client versus the server. In other words, if you maintain state on the server (i.e. STATEFUL session bean), the server will be taxed more unlike with STATELESS bean where the client manages state.

Despite their differences, session and entity beans do share some of the following characteristics:

- A handle (javax.ejb.Handle) can be obtained to a bean's instance using the getHandle() method.

- A bean can be customized at deployment time by editing its environment properties.

- Instances of a bean are created and managed at runtime by the application server.

- Certain attributes of the bean, such as transaction mode and security, can be manipulated at deployment time by using the EJB server deployment tools.

- They can be transaction aware.

- The beans can be created and deleted (destroyed) using the createXXX and *remove* methods using their home interface. The create methods are used to typically initialize the object's fields.

Figure 2.6 portrays a Hypothetical snapshot of an EJB server environment to convey the following points:

- An EJB Server can have multiple containers.

- A container can have multiple beans of different types.

- An entity bean maps to a row in a database.

- A client can invoke any type of bean. Stateful session beans are tied to a specific client.

- Stateless session beans are NOT tied to any client and used from a pool as necessary.

- An entity bean can be shared across clients and accessed concurrently.
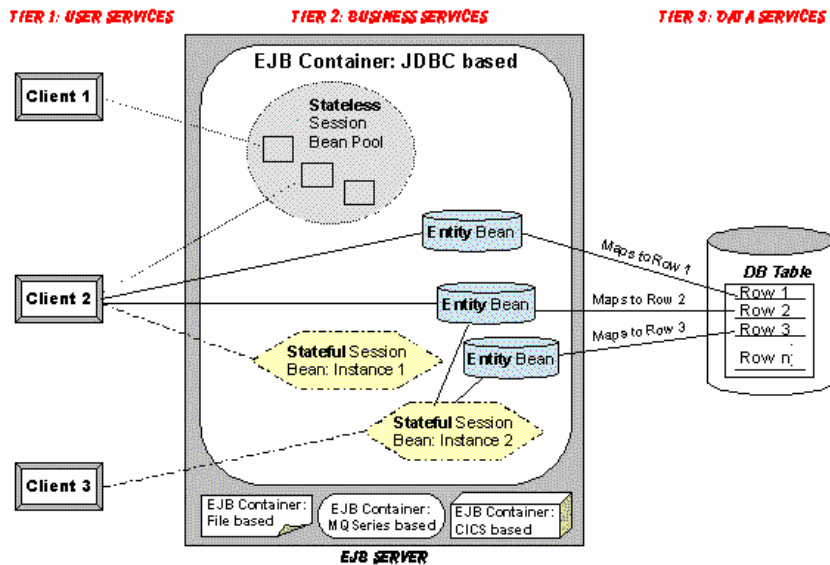

Figure 2.6 Interaction across Clients, Beans and Tiers

Persistence

One of EJB's key features is built-in persistence by the way of *Entity Beans*. The persistence can either be "Bean managed" or "Container managed." For bean managed persistence, the developer overrides the ejbLoad() and ejbStore() methods and provides the custom code (for example, by using JDBC) to transfer the data from the object's fields to the permanent store and vice versa. While bean managed persistence provides a lot of flexibility, it ties the entity bean to one method of persistence, such as JDBC, and requires the developer to write and maintain the persistence code.

Transaction Management

EJB supports flat transactions, modeled after the CORBA Object Transaction Service (OTS) version 1.1. For transactions, no distinction is made between a session and entity bean except for one subtle difference that only allows stateful session beans to suspend a user managed transaction (when using the TX_BEAN_MANAGED transaction attribute, dicsussed later in this article). Additionally, a developer is not exposed to the complexity of transactions which could potentially be distributed across multiple data sources on multiple platforms – this responsibility is shifted to the Java middleware (or app server) provider. Transaction demarcation in an EJB environment can be handled in one of two ways. First, a developer can programmatically control the scope of transaction using the javax.jts.UserTransaction interface (something an app server vendor must provide an implementation for) at the client or server. Secondly, the app server provider manages the transaction boundaries using the deployment settings for the transaction attribute which can either apply to the entire bean or it's specific methods (note: this method is also known as declarative or attribute-based programming). Table 2.2 provides a list of valid values for the EJB transaction attribute – these can typically be set for a given bean using tools provided by the app server vendor:

Attribute Value	How Enterprise Bean is Invoked
TX_NOT_SUPPORTED	Invoked without a transaction scope.
TX_BEAN_MANAGED	Enterprise Bean can use the *javax.jts.User Transaction* interface to demarcate transaction boundaries. The *UserTransaction* interface can be obtained using the *javax.ejb.Session Context.getUser Transaction()* method. An instance of a stateless session Bean or an entity Bean is not allowed to retain an association with a transaction across multiple calls from a client.
TX_REQUIRED	If client is associated with a transaction context, then the enterprise bean is invoked in same context, otherwise the

	container starts a new transaction before invoking a method on the bean and commits the transaction upon returning from the method.
TX_SUPPORTS	Invoked in the client's transaction scope, if client has one. Otherwise, invoked without a transaction context.
TX_REQUIRES_NEW	Always invoked in a new transaction.
TX_MANDATORY	Always invoked in the scope of the client's transaction. If client does not have one, *javax.transaction.TransactionRequiredException* is thrown to client.

Table 2.2: Valid values for transaction attribute

EJB also provides support for Transaction Isolation Levels; a concept, which makes changes, made by other transactions visible to certain transactions. The valid values for EJB isolation levels, also set in the deployment descriptor typically using vendor tools, are as follows:

- TRANSACTION_READ_UNCOMMITTED

- TRANSACTION_READ_COMMITTED

- TRANSACTION_REPEATABLE_READ

- TRANSACTION_SERIALIZABLE

For session beans and entity beans with bean managed persistence, the specified isolation level is set in the database connection (for example, the JDBC isolation levels). For container managed entity beans, the vendor is responsible for providing this functionality.

Relationship to JDBC 2.0

JDBC 2.0 introduces standard extensions API (javax.sql package) which includes support for distributed transactions in addition to features such as resource pooling, rowsets and more. The support for distributed transactions allows developers to write enterprise beans that are transactional across multiple DBMS servers. In order to provide a standard way of supporting distributed transactions in Java, JDBC ties into the Java Transaction (JTA) APIs and Java Transaction Service (JTS). The JTS API provides the necessary interfaces and classes for transaction management in Java. One of it's features is the ability to attach a JDBC driver to an external transaction manager using the standard X/Open XA interface (JDBC 2.0 introduces the javax.sql.XADataSource and javax.sql.XAConnection interfaces, which any drivers supporting distributed transactions must implement, if the driver vendor claims to support the standard JDBC extensions).

Exception Handling

The EJB 1.0 specifications define two types of exceptions, Application and System level exceptions. Since a client accesses a bean's methods via its Home and Remote interfaces, all methods defined in these interfaces must include a java.rmi.RemoteException. When a RemoteExeption is thrown, a client can assume that it is a system level failure. All other exceptions, including the javax.ejb.CreateException, javax.ejb.Remove Exception, and javax.ejb.FindException are considered application level failures. In relationship to transactions, a client can assume that a transaction was rolled back if a javax.jts.TransactionRolledbackException exception is thrown (note: the javax.ejb.SessionContext.getRollback Only() method can also be called to test if a transaction was rolled back using the setRollbackOnly() method).

Security

EJB uses the java.security.Identity class to describe users and/or roles. The app server (or EJB container) actually performs the mapping, usually in a platform-specific way, and provides this information to the bean instance via the getCallerIdentity and isCallerInRole(Identity ident) methods in the javax.ejb.EJBContext class. EJB security uses Access Control Lists in the deployment descriptor to manage security on behalf of the bean. EJB also provides attributes, RunAsMode and RunAsIdentity, to define the security Identity to be associated with the execution of methods in the bean and any calls to other resource managers (e.g. JDBC). The valid values for these attributes are SPECIFIED_IDENTITY, CLIENT_IDENTITY or SYSTEM_IDENTITY (privileged account).

Naming and Directory Service

EJB uses the Java Naming and Directory Interface (JNDI) for it's naming service. In other words, a client looks up an EJB's home interface using JNDI. It is the container's responsibilitiy to make the EJB component's classes available to the client via JNDI.

Wire Protocol

The EJB specifications by default specify that Java RMI Protocol (JRMP) as the default protocol for invoking EJB components over the network. Additionally, the specifications provide CORBA/IIOP mapping so CORBA clients can invoke EJB components. However, EJB

is not in any way limited to JRMP and IIOP, in fact, it could be any other distributed protocol (e.g. HTTP, DCOM). Either way, the developer doesn't have to worry about the protocol as the EJB vendors handle the low level details. Figure 2.7 shows the various ways EJB components can be accessed; for example, the following sample architecture patterns are depicted in figure 2.7:
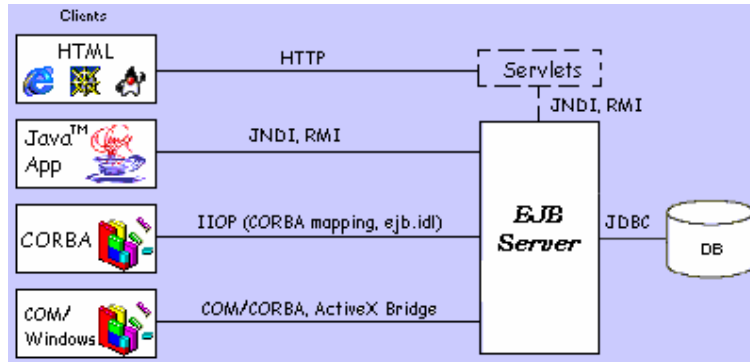
Figure 2.7: Sample EJB based Architecture Patterns and Corresponding Protocols

Additionally, a Java middleware could provide support for other protocols such as DCOM or HTTP, to invoke EJB components. While the default wire protocol for EJB is the Java RMI Protocol, the EJB 1.0 specifications define a CORBA mapping. Essentially, this is accomplished via the IDL files (e.g. ejb.idl) provided for the EJB classes and interfaces combined with an application specific IDL for the bean specific classes and interfaces. The mapping for CORBA also covers propogation of the transaction, naming and security contexts as defined by the CORBA Object Services protocols.

Attribute Based Development

EJB not only enhances a developer's productivity by providing a simple server side component architecture but also by introducing declarative, or attribute-based, programming. For example, attributes for transactions, isolation levels, security, and mapping to EJB container data sources can all be handled at deployment time. Hence, the developer does not have to worry about these low-level details when developing their components, instead they can simply concentrate on the business logic.

14.6 Deployment

EJB components are deployed via a Java ARchive (JAR) file. A JAR file is a nice and neat way to include the following, required and optional, files:

- Enterprise Bean's class files (home interface, remote interface and actual bean)

- A deployment descriptor for each bean. A deployment descriptor is a serialized instance of a javax.ejb.deployment.EntityDescriptor or javax.ejb.deployment.SessionDescriptor object.

- Manifest files that list the names of the deployment descriptor files and also indicate which ones are Enterprise Beans. The format of this file looks something like this:

  Name: divya/infoBookServer.ser

  Enterprise-Bean: True

  Name: xyz/QuoteServer.ser

  Enterprise-Bean: True

- Environment properties for the bean (typically entered using vendor tools).

14.7 Roles and Responsibilities

Sun's EJB v1.0 specifications define six different roles or people that can be involved in development and deployment of EJBeans (see Table 2.3). Of course, this will vary based on the size of your project and/or company. For example, in a small project, the same person could be the Bean Provider, Application Assembler, Deployer, and System Administrator. Similarly, one vendor could be the EJB Server and Container Provider (for example, Information Builders provide several EJB containers with their EJB compliant application server).

Role	Responsibilities
Bean Provider	• Classes and interfaces • Environment Properties • Deployment Descriptor • Manifest file • EJB Packaging (JAR file)
Application Assembler	• Brings all tiers together (e.g. client, GUI, servlets, etc.)
Deployer	• Uses container tools to map bean to container (if needed) • Modifies security attributes (if needed) • Bean's properties (if needed)
EJB Server Provider	• Could be OS, middleware, or database vendor • Provides container for session beans • Possibly provides containers for entity beans or publishes it's low level interfaces to allow third party plug-in containers
EJB Container Provider	• Responsible for persistence of entity beans • Mapping Tools for entity beans (e.g. Object/relational mapping) • Generates code to move data from entity bean instance variables to secondary storage • Provide tools to manage container and beans running in that container
System Administrator	• Monitor system • Fine tune (if needed)

Table 2.3 Roles and Responsibilities

14.8 Short summary

✎ A HTML frontend could invoke a servlet which in turn invokes EJB components…

✎ A Java applet or stand-alone application can invoke the EJB components directly.

✎ One of EJB's key features is built-in persistence by the way of *Entity Beans*.

- EJB enables software developers to build server-side, reusable business objects.

- **Entity Bean** is persistent objects and represents an object view of data stored in permanent storage.

- **Stateful Session Bean** provides easy and transparent state management on the server side.

14.9 Brain Storm

1. What is EJB?
2. What is the Need of EJB?
3. Explain the Architecture of EJB.
4. What are the Features of EJB?
5. What are the features of EJB?
6. What is JNDI?

ಖಿಂ

Lecture 15

Creating a Simple Enterprise JavaBean

Objectives

In this lecture you will learn the following

- ✎ How to Design EJB?

- ✎ About Architecture

- ✎ About Implementation

- ✎ Looking into the working of EJB

Coverage Plan

Lecture 15 &16

15.1 Snap shot

This lecture will discuss on Creating a Simple Enterprise JavaBean, Architecture, Implementation And the working the working methodology of EJB.

15.2 Requirement

We build an example that illustrates the process of building, deploying and using an Enterprise JavaBean. The Client will call a simple method on an Enterprise JavaBean.

15.3 Design

For the Server side object, we need one method that takes an argument and returns a result. For this "HelloWorld" example we'll name the method say hello(), and its method signature will be as follows.

public String sayHello(String s)

The design of the client is as follows.

1. Find the Bean's home Interface.
2. Get access to the Bean.
3. Call the Bean's sayHello() method with a string as argument and save the
   return value.
4. Display the return value to the user.

15.4 Architecture
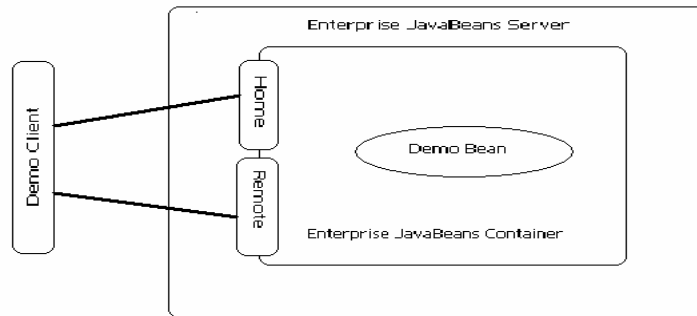
The home and Remote Interfaces are created at Deployment time.

15.5 Implementation

Step 1: Create the Remote Interface for the Bean

A remote is the with which the client interacts. The EJB container uses the remote interface for your bean to generate both the client-side stub and a server-side proxy object that passes client calls to your EJB object. In EJB, the client is never allowed to obtain a direct reference to your EJB implementations; access to the implementation is always brokered the container.

Listing 2.1 shows the code for the bean's remote interface.

```
1.  package ejbeans.progs.first;
2.  import javax.ejb.*;
3.  import java.rmi.*;
4.
5.  public interface Hello extends EJBObject
6.  {
7.  String sayHello(String s) throws RemoteException;
8.  }
```

Listing 2.1

The first line is a package declaration; it specifies that the Hello class will be part of the ejbeans.progs.first.

In Line 2 OF Listing 2.1, all classes in the package javax.ejb are imported. This package is prefixed with "javax" rather that with "java" because it is an extension to Java, and not a part of the core JDK packages.

In line 3, the classes in the java.rmi package are imported. Note that the java.rmi package is included as part of the standard JDK package. This package is imported because line6 references the java.rmi.RemoteException.

Line 5, declares the public interface Hello as extending the javax.ejb.EJBObject interface. At deployment time, the container must generate a class that implements the Hello interface. The container-generated class passes method invocations to the EJB implementation class and allows the client to perform the following tasks:

- Get a reference to the bean's home interface (via the getEJBHome() method).

- Get a reference that can be saved and then restored at a later time(via the getHandle() method).

- Get rid of the object (via the remove() method).

- Test whether two remote objects are identical (via the isIdentical() method).

Line 7, supplies the method signature of the one business method. This exception is straightforward; it takes a String as an argument and returns a string. All arguments and return types for EJB business methods follow the same rules that are required for RMI; that is, they must be primitive types, be serializable, or be a remote reference. This situation occurs because the arguments and return vales must be serialized so as to pass over the wire from the client to the server.

Note: The business method throws a java.rmi.RemoteException. This exception can be thrown at any time if a network problem occurs on the connection between the client and the server. Because network outages can arise at any time, a client should be prepared to handle a RemoteException gracefully. All business methods must include the java.rmi.RemoteException exception in their "throws" clause. (You may choose to throw other exceptions in your business methods, but you must include the RemoteException.) The implementation for the remote interface is created by the container; the EJB developer required to supply only the interface.

Step:2 Create the Bean's Home Interface

The next step in building an EJB application is to create a home interface for your EJB implementation. The home interface is the client's initial point of contact with your EJB components. The client obtains a reference to an object implementing the home interface via JNDI. After the client has obtained this reference, it can use a create() method to obtain a reference to an object that implements the remote interface discussed in step 1.

```
1. package ejbeans.progs.first;
2. import javax.ejb.*;
3. public interface HelloHome extends EJBHome
4. {
5. public Hello create() throws
6. java.rmi.RemoteException,javax.ejb.CreateException;
7. }
```

Listing 2.2

Line 1 declares that this class will be part of the ejbeans.progs.first. In line 2, we again import the class es in the javax.ejb package. Line 3 declares the home interface HelloHome as extending the EJBHome interface.

In line 6, declare the only create() method. The create methods essentially replace constructors in EJB as the means of initializing an object, thereby allowing the container implementers to avoid the overhead of instating a new object each time a client requests a reference. The container can maintain a cache of already-instantiated objects and use the create() call to initialize the internal state of the object; this process is speedier than instantiating and initialising object from scratch. Some particulars about create() methods follow:

- At least one create() method must be supplied in a Session bean's home interface.

- The create() method must be the public, because it will be called from outside its package.

- The return type must be the type of the remote interface.

- The "throws" clause may also include other application-specific exceptions.

The javax.ejb.CreateException notifies the client that a new EJB object could not be created. This exception may be thrown by the EJB object itself or by the container. As with the remote interface, note that the EJB developer is not responsible for providing an implementation of the home interface. Instead, the EJB container generates the implementation class at deployment time.

Step: 3 create The Bean's Implementation Class

This step shows you how the code of the application(business logic) is done. So far we have declared interfaces that the container tools will generate the code. But here is where the functionality of the JavaBean is coded.

```
1.  package ejbeans.progs.first;
2.  import javax.ejb.*;
3.  public class HelloBean implements SessionBean
4.  {
5.  SessionContext ctx;
6.  public void ejbCreate( )
7.  {
8.  }
9.
10. public String sayHello(String s)
11. {
12. return "Hello there,"+s;
13. }
14.
15. public void ejbRemove( )
16. {
17. }
18. public void ejbPassivate( )
19. {
20. }
21. public void ejbActivate( )
22. {
23. }
24. public void setSessionContext(SessionContext  ctx)
25. {
26. this.ctx=ctx;
```

```
27. }
28. }
```

Listing 2.3

Line 1 of listing 2.3 declares that this class is part of the ejbeans.progs.first. In line 2, we import the classes in the javax.ejb interface. Line 4 declares that the implementation, HelloBean, implements the SessionBean interface. The SessionBean interface contains the following method signatures:

public abstract void ejbActivate();

public abstract void ejbPassivate();

public abstract void ejbRemove();

public abstract void setSessionContext(SessionContext ctx);

Because we are using a Session bean, it implements the SessionBean interface. Entity beans (naturally) implement the EntityBean interface. We'll explain the function of each of these methods in due course. For now, note that because we have declared these methods in the SessionBean interface, we must provide an implementation of them, even if that implementation is merely an empty method.

Line 6 declares a class variable to hold a reference to a SessionContext variable. The bean uses the SessionContext to interact with the container.

> **Note:** Why does a stateless bean have a member variable?
> In fact, the SessionContext is not a normal member variable. The life cycle of a stateless Session bean has two states: either the bean doesn't exist or it's ready to receive method calls. Each time you create a stateless Session bean, its SetSessionContext() method is invoked. Thus, because the container always invokes setSessionContext() on the bean before it's added to the pool, the stateless Session bean is assured of always having a valid reference to a SessionContext.

It is known that the SessionContext variable is set only once, when the bean is created

But it is also said that a stateless Session bean can be invoked by many different clients.

So, if the SessionContext is set only once, how can the bean have current information in the SessionContext ? .

The reference that's passed in setSessionContext() is not simply a static object; rather its an interface reference that allows the EJB class to query its container for the current state. The container is responsible for making sure that when the bean calls getUserTransaction() on its SessionContext(), the transaction that is returned applies to the current client.

Line 7 contains the declaration for the ejbCreate() method. This ejbCreate() method corresponds to the create() method declared in the home interface in step 2. When the container generates an implementation for the home interface, it calls a bean's ejbCreate() method for each corresponding create() method in the home interface. The EJB developer must write these ejbCreate() methods. Each ejbCreate() method must follow several rules:

- It must be declared public.

- It must have a void return type. Even though the create() method returns an object that implements the remote interface, the actual creation of this object is the responsibility of the container.

- The number and type of arguments in an ejbCreate() method must mirror the number and type of arguments in the create() method.

- The ejbCreate() method is not required to throw any exceptions. Its' also possible to throw arbitary exceptions from within an ejbCreate() method. In such a case, they must also be declared in the "throws" clause of the corresponding create() method.

Even though the class has no variables that must be initialized within the ejbCreate() method, we must still provide an implementation of it. In fact, a stateless Session bean must have exactly one ejbCreate() method, and it must take no arguments.

Line 11 contains the method which holds the business logic. This method signature exactly matches the one declared in the remote interface back in step 1, except that it does not throw a java.rmi.RemoteException. An EJB class can have any number of business methods, as long as each business method has a corresponding declaration in the remote interface. Line 16 contains an empty implementation of the ejbRemove() method.

The container calls this method to notify the EJB instance that it will soon be removed. This strategy allows the instance to take care of any last-minute housekeeping (closing

files,writing data, committing transactions) before it passes out of existence. Line 19 contains an empty implementation of the ejbPassivate() method. The EJB container can swap out the EJB instances to temporary storage if necessary. Before the container swaps the Session bean out to temporary storage, it invokes the bean's ejbPassivate() method. The bean then has an oppurtunity to free up any references it may be using (for example, if it has an open socket connection, it must close the socket before being passivated). Line 22 contains an empty implementation of the ejbActivate() method which is also called by the container. The ejbActivate() method is the inverse of the ejbPassivate() method; the container calls it just after the Session bean has been read back in from any temporary storage.

This method gives the bean the opportunity to restore any connection that it may need before going back into service. Line 24 implements the setSessionContext() method of the SessionBean interface. Remember that the SessionContext interface is a special case; it is the only object variable that a stateless Session bean will normally have. A stateless bean could have other information stored in object variables; for example, a sales tax bean would need to keep a database reference to perform its lookups. The key thing to remember with stateless Session beans is that you should not store conversational state in your beans, because your client does not have a long-term relationship with any particular bean.)

Step 4: Compile the Remote Interface, Home Interface, and Implementation Class

```
javac *.java
```

Step 5:Create A Session Descriptor

There are different ways to create a Session Descriptor.

The SessionDescriptor class passes information about your bean to the eventual deployment environment. At development time, the bean developer creates an instance of the SessionDescriptor class, fills it with information, and then serializes it. At deployment time, the developer deserializes this instance and uses the information it contains to deploy the bean. Typically, the vendor of your EJB development environment will provide some type of tool for generating descriptors. You must set a few key values in the SessionDescriptor class for the HelloBean:

- Make sure you're creating a SessionDescriptor, and not an EntityDescriptor. Session and Entity beans each have their own descriptor.

- Set the BeanHomeName property to HelloHome. The JNDI naming service will then associate this name with ythe home interface. Clients will look this name up via JNDI and be presented with a reference to an object implementing your home interface. You can pick any name you like – you could name it Fred if you wanted to. The name you associate with your bean in JNDI need not map to a particular class name in your bean.

- Set the EnterpriseBeanClassName property to ejbeans.progs.first.HelloBean.
- Set the HomeInterfaceClassName to ejbeans.progs.first.HelloHome.

- Set the RemoteInterfaceClassName to ejbeans.progs.first.Hello.

- Set the StateManagementType property to STATELESS_SESSION.

As noted earlier, your EJB development environment will likely supply some sort of tool to generate a serialized deployment descriptor. BEA Web-Xpress's WebLogic product, for example, supplies the class weblogic.ejb.utils.DDCreator. This class can be run from the command line.

```
java.weblogic.ejb.utils.DDCreator  filename
```

where <file name> is the name of the file containing the information that your SessionDescriptor or EntityDescriptor should contain. The easiest way to create a new descriptor file is to start with an old one and modify it to suit your needs.

```
java.weblogic.ejb.utils.DDCreator  - example  >
filename
```

The DDCreator utility will then spit out a sample file in the proper format. Taking this file as a starting point, simply adjust the various properties until they reflect your desired attributes, and then run the DDCreator utility on the file to create a serialized deployment descriptor that contains the appropriate information.

Below is a sample descriptor file for which you can change the settings to your own classes.

Listing 5.1 – Session Descriptor file – f.txt

(SessionDescriptor; This file must start with SessionDescriptor or EntityDescriptor ;

Indicate the name which the bean will be bound into the JNDI name as

```
beanHomeName                first.HelloHome
; The enterprise Java Bean class (see step #4)
enterpriseBeanClassName     ejbeans.progs.first.HelloBean

homeInterfaceClassName          ejbeans.progs.first.HelloHome
; The home interface implemented by a class generated
  by the

container
; provided tools see step #3

remoteInterfaceClassName        ejbeans.progs.first.Hello

; See step #2

  isReentrant                     false

; Always false for session beans

  stateManagementType            STATELESS_SESSION
  ; Either STATELESS_SESSION or STATEFUL_SESSION.
  ; DemoBean is a stateless session bean

  sessionTimeout                 5; seconds

  (controlDescriptors
  ; This section decides the run-time properties when a method is called.
  ; The DEFAULT sub-section applies to all methods,  but can be overridden
  ; on a per-method basis, similar to the "accessControlEntries" above.
    (DEFAULT
isolationLevel              TRANSACTION_SERIALIZABLE
```

```
        transactionAttribute        TX_REQUIRED



        runAsMode                   CLIENT_IDENTITY

    ); end isolationLevel


  ); end controlDescriptors


  (environmentProperties


   maxBeansInFreePool            100


  ); end environmentProperties
); end SessionDescriptor


Use DDCreator utility..
```

```
    java.weblogic.ejb.utils.DDCreator  f.txt
```

This will create a serialized descriptor file called helloBeanDD.ser

You can create your own descriptors by writing a Java program to fill in the values of a SessionDescriptor class and then serialize the class. As given below.

//Java Programming to Provide Values for and Serialize a //SessionDecsriptor //Class

```
1.      import javax.ejb.deployment.*;
2.      import java.io.*;
3.      import java.util.Properties;
4.
5.      public class DDwrite
6.      {
7.      public static void main (String argv[ ])
8.      {
9.      System.out.println("This      program      creates     a      serialized
        deployment descriptor");
10.     SessionDescriptor sd = new SessionDescriptor( );
11.     //set bean class name
12.     sd.setEnterpriseBeanClassName("ejbeans.progs.first. Hel loBean");
13.     //set home interface name
14.     sd.setHomeInterfaceClassName("ejbeans.progs.first. Hell oHome");
```

```
15.    //set remote interface class name
16.    sd.setRemoteInterfaceClassName("ejbeans.progs.first. Hello");
17.    //set environment properties
18.    Properties p =  new Properties ( );
19.    p.put("myprop1", "myval");
20.    sd.setEnvironmentProperties(p);
21.    //Session-specific entries
22.    //session timeout -- value in second (0 means use
23.    container-specific timeout);
24.    sd.setSessionTimeout(0);
25.    //state management type -- stateless or stateful
26.    sd.setStateManagementType(SessionDescriptor.STATELESS_
       SESSION);
27.    try
28.    {
29.    FileOutputStream fos =new
30.    FileOutputStream("foo.ser");
31.    ObjectOutputStream oos = new ObjectOutputStream(fos);
32.    oos.writeObject(sd);
33.    oos.close( );
34.    }
35.    catch  (IOException ioe) {
36.    System.out.println("Error writing serialized
37.    deployment  descriptor: " + ioe);
38.    }
39.    }
40.    }
41.    }
```

In line 10, the program creates a new instance of the SessionDescriptor class. In lines 12-18, the program sets the name of the bean's class and the names of its home and remote interfaces. At deployment time, the vendor's tools can use Class.forName() to load instances of these classes, and can then use introspection on these classes to determine their method names.

Lines 20-24 creates a java.util.Properties object, and set the property myprop1 to the value myval. The EJB container must make these properties available to your beans at runtime and should also allow you to edit them during the deployment process. Properties are a useful mechanism for encapsulating any outside references that your bean may require at runtime. For example, if your bean must obtain a reference to another bean, it will need to use JNDI (or

CORBA COS naming; but let's stick to JNDI for the moment). To get a JNDI InitialContext, you must tell JNDI where to obtain an initial context. You accomplish this task by setting the environment property java.naming.factory.initial to point to a suitable InitialContextFactory. (A Factory is a class whose sole purpose is to create and return instances of another class. The factory idiom is often used in the JDK specification; for example, java.rmi.server. RMISocketFactory is used to create RMI sockets, and the Swing package includes several factory classes for the creation of graphical objects). Setting the java.naming.factory.initial property allows you to avoid hard-coding a reference to the initial context factory, which allows for more flexibility in deployment.

Line 28 sets the timeout value for the session bean to 0, thereby instructing the container to use its default value for timeouts. Line 31 instructs the container that this bean is a stateless or stateful, because the life cycles of stateless and stateful beans differ. Session bean life cycles are examined in detail later in chapter 4; for now its' sufficient to say that a stateless bean has a much simpler life cycle than a stateful bean. If the container wants to swap out a stateful bean, it must serialize the bean to some type of persistent store, so that the bean can be reinitialized when it is needed again. On the other hand, if the container needs to discard a stateless Session bean, it can simply get rid of it.

Lines 33-43 create the serialized deployment descriptor. Line 35 opens a file output stream to a file named foo.ser – this file will contain the serialized deployment descriptor. Line 36 wraps the FileOutputStream object with an ObjectOutputStream object. Line 37, invokes the ObjectOutputStream's writeObject() method, which performs the actual serialization. Line 38 closes the file and the program then exits. After running this program, you should find a file named foo.ser on your disk (as described in the Outputstream). This file should contain a serialized deployment descriptor.

Step 6: Create A Manifest

 A manifest file is a text document that contains information about the contents of a jar file. Actually, the jar utility will automatically create a manifest file even if none exists.

```
Name: ejbeans/progs/first/HelloBeanDD.ser
Enterprise-Bean: True
```

Step 7: Create An EJB –Jar File

Use the jar program that comes with the Java JDK to generate your ejb-jar file. The invocation looks much like the following line:

```
jar cmf ejbeans\progs\first\manifest hello.jar
ejbeans\progs\first\*.class
ejbeans\progs\first\ *.ser
```

```
where
jar = the program name
cmf = the options (c – create;m – include manifest;f – file
name)
ejbeans\progs\first\manifest = the path from the current
directoru\y to the manifest
hello.jar – the name of the jar file to deploy.
ejbeans\progs\first\- the directory to include in the jar file.
```

The end result is an ejb-jar file suitable for deployment.

Step 8: Deploy the EJB-Jar File

The deployment process will vary slightly, depending on which EJB implementation you use. We'll use BEA WebXpress's WebLogic application server here, so we'll describe the steps taken to deploy a bean in WebLogic. For specific instructions on how to deploy a bean in a particular EJB implementation, consult your vendor's documentation. Typically, the vendor will provide some type of deployment tool that examines either the jar file itself or the serialized deployment descriptor contained therein, generates any additional classes required to run the bean (for example, implementations of the home and remote interfaces) and installs the bean in the server's class path. In WebLogic, the tool used to deploy a bean is a Java class called weblogic.ejbc ("ejbc" stands for EJB compiler). This program takes the name of a serialized deployment descriptor as argument and performs the following steps:

- Creating additional classes, such as the implementation classes for the home and remote interfaces.

- Moving your EJB classes to the appropriate directory in the WebLogic servers class path.

After running ejbc on your deployment descriptor, you must undertake one final step to deploy your beans in WebLogic. In the top-level directory in which you installed WebLogic (Normally, this directory is c:\weblogic), you'll find a file called weblogic.properties. It contains information and directives that the WebLogic server reads when it is started up. Set the property weblogic.ejb.deploy to point to your deployed jar file. The weblogic.ejb.deploy property should already exist in the file, but it is initially commented out; simply do a text search for it, uncomment it (remove the "#" character at the beginning of the line) and add a reference to your ejb-jar file to it. After you've modified the weblogic.ejb.deploy property to include a reference to your ejb-jar file, start the server (say t3server on the command line). The server should then load your EJB and make it ready for user.

Step 9: Write A Client

```
1.      package ejbeans.progs.first;
2.      import ejbeans.progs.first.*;
3.      import java.rmi.*;
4.      import javax.ejb.CreateException;
5.      import javax.naming.*;
6.      public class HelloClient{
7.      public static void main (String[ ]   args){
8.      try
9.      {
10.     Context ic = new InitialContext( );
11.     HelloHome home =(HelloHome) ic .lookup("HelloHome");
12.     Hello hel = home.create( );
13.     String retval = hel.sayHello("To whomever you are");
14.     System.out.println("returned:" +retval);
15.     hel.remove( );
16.     }
17.     catch (java.rmi.RemoteException  e)
18.     {
19.     System.out.println("remote exeception  occurred: "
        +e);
20.     }
21.     catch  (javax.ejb.CreateException  e)
22.     {
23.     System.out.println("create exception occurred: " +e);
24.     }
25.     catch (javax.ejb.RemoveException  e)
26.     {
```

```
27.    System.out.println("remove exception occurred :"+ e);
28.    }
29.    catch (javax.naming.NamingException  e)
30.    {
31.    System.out.println("naming exception occurred:"+ e);
32.    }
33.    }
34.    }
```

Line 3 imports the JNDI classes. This client uses JNDI to look up a reference to our EJB object's home interface, so we must include this package. In line 11, we create a class of type javax.naming.InitialContext(). The InitialContext class serves as the client's interface to JNDI; it can contain bindings to a variety of naming services (JNDI, CORBA, COS,DNS, and so on). The InitialContext class provides the client with a single interface that can be used to link to any naming services in the client's environment that support JNDI. This statement can throw a javax.naming.NamingException if it cannot complete; this exception is caught in the catch block in lines 30-33. Line 12 uses the InitialContext class to perform a lookup on the name HelloHome (the name of the home interface in this example).

 JNDI searches its namespace for an occurrence of this name and returns the object to which this name has been bound. In this case, the object is a generated object that implements the HelloHome interface. One-of the container's deployement-time task is to create a JNDI name that is bound to an implementation of the Bean's home interface. The Bean developer includes such a name in the deployment descriptor, but the deployer has the final say as to what the interface's JNDI name will be. Because the InitialContext. lookup() call returns an object of type Object, we must cast it into a variable that can hold a reference to the home interface before using it.   Line 13 invokes the create() method on the home interface to obtain a reference to an object that implements the Bean's remote interface.

The EJB container generates this object as part of the deployment process. After invoking the create() method, we can then invoke any necessary business methods. In line 14, we invoke the sole business method, sayHello(), with the argument string "Whom ever You are", and receive the return value of this method. Now that we have finished the Bean, we use the remove() method in line 16 to tell container to discard it. Session beans will eventually go away on their own, either as a result of their timeout period expiring or as a result of a server crash or shutdown. Nevertheless, the best strategy is to invoke the remove() method of an object when you no longer need it; this approach allows the server to free up the session Bean

and any of its associates resources. Following the logic are a number of catch blocks to handle any exceptions that might be thrown in the process of looking up, creating, conversing with, and removing the remote Bean. Line 18-21 catches the java.rmi.RemoteException. Recall that this exception can be thrown at any time during a remote conversation; typically, it indicates that some sort of communication problem has occured between your client and the server. Line 22-25 catches the javax.ejb.CreateException. Either the Bean or the container can throw this exception if a problem occurs during the ceation of a Bean.The javax.ejb.RemoteException handled in line 26-30 is a similar exception; it can be thrown either by the Bean or by the container if a problem occurs during the removal of a Bean. Line 30-33 handles the javax.naming.NamingException: this exception can be thrown during the creation of the JNDI. InitialContext or during the lookup step.

Step 10: Run the Client

JNDI will need to know where it can find an InitialContextFactory.initial. It finds this information in the environment property java.naming.factory.initial.In this example, we simply pass a value for this property as a command-line argument. Here's the invocation of the client:

```
java -Djava.naming.factory.initial=weblogic.jndi.T3InitialContextFactory HelloClient
```

Note that your client must be able to find the following classes in its class path:

- Your Bean's home interface
- Your Bean's remote interface
- The javax.naming.InitialContext class
- The stub classes generated by the deployment process to handle communication You have several options for making these classes available to your client program:
- You can include them with your code and write an installer routine to set everything up on the client.
- In an Intranet environment, you can establish the application and its associated classes on a shared file server's directories in their CLASSPATH.
- If you've written an applet, you can simply include the classes as part of the CODEBASE.

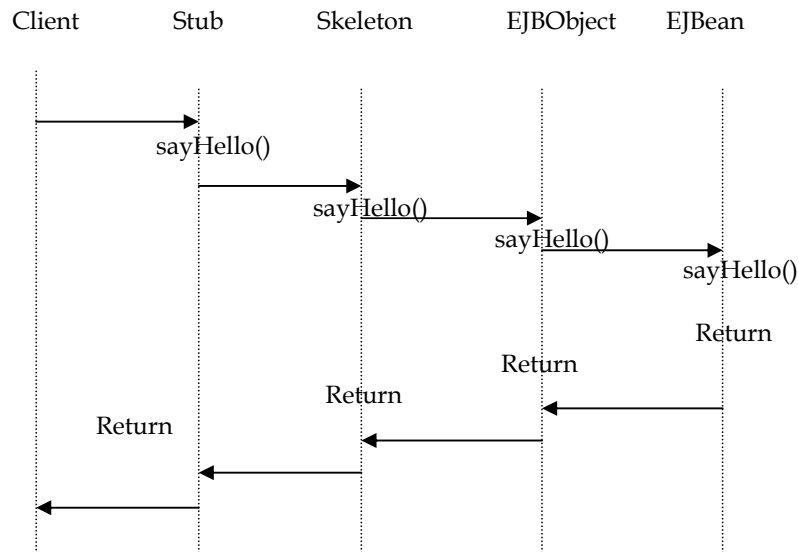## 15.6 Looking into the working

To this point, we have discussed communication between the client and the server at a fairly high level. Now let's move to a somewhat lower level and discuss what really happens when you call a method on a remote object? The diagram in Figure 2.8 shows the entities involved in an EJB conversation as well as the role that each entity plays.

1. In your client program, you invoke a remote method on a remote interface.

2. The method invocation goes to a client-side stub (a proxy object that implements the remote interface), which serializes the arguments and sends them over the wire.

Note: The process of serializing the arguments into a form suitable for transmission over the network is called marshalling. This marshalling process has several implications:

- Any variables that you pass arguments to a remote object must obey the rules for RMI serialization. Additionally, any classes contained within those classes obey the same rules. Breaking these rules will cause an exception to be thrown.
- Because all argument classes and their contents must be serialized and sent over the wire, it's best to keep these classes as small and simple as possible. Serializing and passing a large object remotely can be an expensive process in terms of time and resources.

---

3. The arguments are received by a skeleton on the remote host, which deserializes the argument and passes them to the EJBObject. The skeleton is the server-side counterpart of the stub; it is usually generated automatically during the deployment process. It receives the stream of bytes generated by the client-side stub and transforms them back into Java objects (a process called unmarshalling). The EJBObject here is the class that implements your remote interface; the container generates it as part of the deployment process. Acting as a proxy for client requests, the EJBObject interposes between the client and the EJB object. This approach allows the server to better manage incoming calls to the EJB object and gives the server a place to handle the automated transaction features of EJB.

4. The EJB object, in turn, calls the actual method on your Bean.Your method then does any necessary processing and eventually returns.

5. When the method returns, the return value is passed to the EJBObject, When the EJBObject invokes the Bean by a method call, it receives the return values as it would any normal return value.

Client   Stub   Skeleton   EJBObject   EJBean

sayHello()

sayHello()

sayHello()

sayHello()

Return

Return

Return

Return

6. The EJBObject passes the return value to the skeleton. Again, it receives the return value after the method has finished executing.

7. The skeleton serialzes the return values and sends it over the wire to the stub.

   The stub deserializes the return value (that is, converts it back to a Java object) and returns it to the calling routine.

15.7 Short Summary

The Following are the steps involved in creating a Simple Session EJB.

♦ Create The Bean's Implimentation Class

♦ Complile The Remote Interface,Home Interface,And Implimentation Class

♦ Create a Session Descriptor

♦ Create a Manifest

♦ Create an EJB-Jarfile

♦ Deploy the EJB-Jar File

♦ Write a Client

♦ Run the Client

15.8 Brain Storm

1. What is the Difference between Session and Entity bean?

2. Why are Home and Remote Interfaces used?

3. What do you mean by Connection Pooling?

Lecture17

Remote Method Invocation

Objectives

In this lecture you will learn the following:

- About Distributed Applications

- About RMI

- About RMI Architecture

Coverage Plan

Lecture 17

17.1 Snap shot

Java Remote Method Invocation (RMI) allows you to write distributed objects using Java. This Module describes the benefits of RMI, how you can build application using RMI and how to connect with other Java tools.

RMI provides a simple and direct model for distributed computation with Java objects. These objects can be new Java objects, or can be simple Java wrappers around an existing API. Java embraces the "Write Once, Run Anywhere model. RMI extends the Java model to be run everywhere."

17.2 Introduction to Distributed Applications

Distributed applications are those that execute across multiple host. Objects executing on one host invoke methods of objects on other hosts. When does this scenario come into play? Take an example where one host, say A, is having an object which performs some special mathematical calculations. Another host, say B, in the same network wants to use this object's method. It invokes the method of the object residing in A by making a remote Invocation making it feel as if it is calling a method of an object in the same (Local) machine. In the above scenario, a notable point is that, B need not have the object to perform the special calculations. This divides the work among various systems. 7

> **Distributed object computing** extends an object-oriented programming system by allowing objects to be distributed across a heterogeneous network, so that each of these distributed object components inter-operate as a unified whole. These objects may be distributed on different computers throughout a network, living within their own address space outside an application, and yet appear as though they were local to an application.
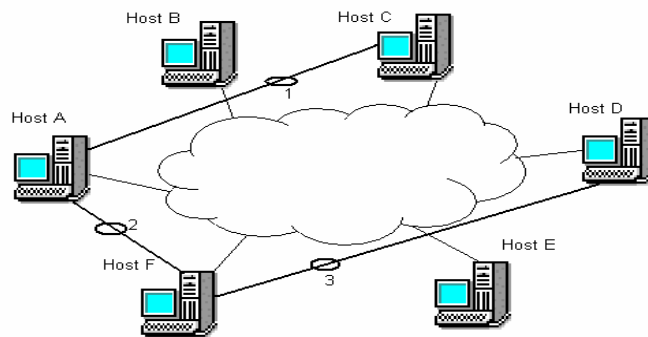
Figure 3.1 – A Distributed Network**.**

In the above figure, the symbol ——◯—— represents a client machine is invoking a method in a remote server machine in the same network. From the figure we can understand the following:

- The Machine named Host A invokes a method of an Object in a remote machine Host C and Host F represented by the Line number 1 & 2 respectively. Here the client is Host A and the Servers are Host C and Host F.

- The machine named Host F invokes a method of an object in a remote machine Host D represented by the Line number 3.

> **Note:** In Client-Server Methodology there is no strict rule that a machine should only be a Client or should only be a Server. For a particular method invocation (request) a machine that serves is said to be the Server, a machine that receives is said to be the Client. The roles of the Client and the Server can be interchanged depending on the invocation. A Server can act as a Client for a particular Invocation and Vice-Versa.

There are a number of approaches to implement distributed systems. The Internet and the Web are examples of distributed systems that have been developed using the TCP/IP Client/Server methodology. Clients and Servers communicate through TCP or UDP sockets. The use of sockets requires separate application-level protocols for Client/Server communication. The overhead associated with these protocols, gives emergence to other approaches to perform distributed computing.

Three of the most popular distributed object paradigms are

- OMG's **Common Object Request Broker Architecture (CORBA)** .
- Microsoft's **Distributed Component Object Model (DCOM).**
- JavaSoft's **Java/Remote Method Invocation (Java/RMI)**.

CORBA relies on a protocol called the **Internet Inter-ORB Protocol (IIOP)** for remoting objects. Everything in the CORBA architecture depends on an **Object Request Broker (ORB)**. The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely. Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. The client can now make method calls on the object reference as if the CORBA server object resided in the client's address space. The ORB

is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client.

DCOM supports remoting objects by running on a protocol called the **Object Remote Procedure Call (ORPC)**. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support *multiple interfaces* each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space.

Java/RMI relies on a protocol called the **Java Remote Method Protocol (JRMP)**. Each Java/RMI Server object defines an interface, which can be used to access the server object outside the current Java Virtual Machine (JVM) and on another machine's JVM. The advantage of RMI over CORBA is that, in order to use CORBA we need to buy the corresponding ORB, but RMI does not need any such ORBs. Similarly, DCOM is not very efficient outside the Windows environment, whereas RMI is platform independent.

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

- A reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote).
- 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.
- 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:

- Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces.
- 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 sing7le virtual machine.
- A remote object is passed by reference, not by copying the actual remote implementation.
- The semantics of some of the methods defined by class **java.lang.Object** are specialized for remote objects.

- Since the failure modes of invoking remote objects are inherently more complicated than those of invoking local objects, clients must deal with additional exceptions that can occur during a remote method invocation.

17.3 Introduction to RMI

Remote Method Invocation (RMI) is the object equivalent of Remote Procedure Calls (RPC). While RPC allows you to call procedures over a network, RMI invokes an object's methods over a network. In the RMI model, the server defines objects that the client can use remotely. The clients can invoke methods of the remote object as if the object was a local object running in the same virtual machine as the client. RMI hides the underlying mechanism of transporting method arguments and return values across the network. In Java-RMI, an argument or return value can be of any primitive Java type or any other Serializable Java objects.

Remote Method Invocation (RMI) is the action of invoking a method of a remote interface on a remote object. A method invocation on a remote object has the same syntax as a method invocation on a local object.

17.4 RMI Architecture

17.4 RMI Architecture

Java RMI is comprised of three layers that support the interface. See illustration   below.
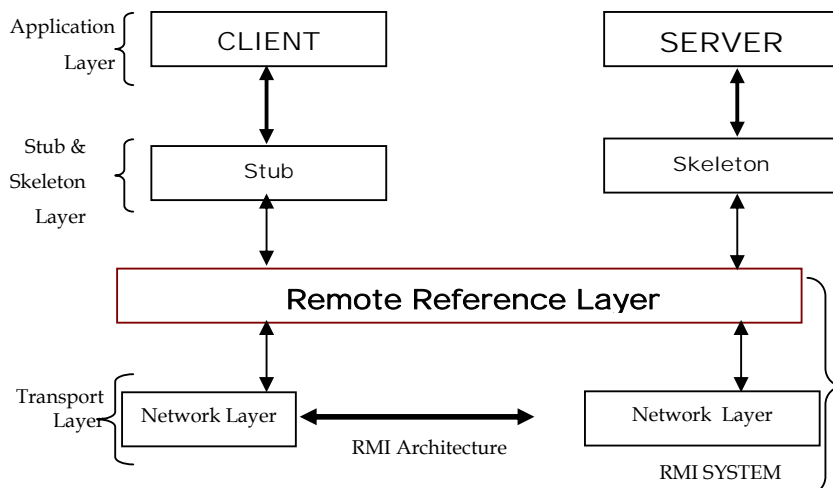

Figure 3.2 RMI Architecture

17.4.1 Application Layer

The Application Layer is where the server makes its methods available for the client to call. This layer is where the end user works.

17.4.2 Stub and Skeleton Layer

The stub and skeleton layer of RMI lies just beneath the view of the application. In this layer, RMI uses the Proxy design pattern. The proxy knows how to forward method calls between the participating objects. In RMI's use of the Proxy pattern, the stub class is the proxy of the remote object, i.e., the server, and the skeleton class is the proxy of the client.

The skeleton is a helper class for RMI. The skeleton understands how to communicate with the stub across the RMI link. The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub.

17.4.3 Remote Reference Layer

The Remote Reference Layer defines and supports the invocation semantics of the RMI connection. This layer provides a RemoteRef object that represents the link to the remote service implementation object.

The stub objects use the invoke() method in RemoteRef to forward the method call. The RemoteRef object understands the invocation semantics for remote services.

Note: The **JDK 1.1** implementation of RMI provides only one way for clients to connect to remote service implementations: a unicast, point-to-point connection. Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system. (If it is the primary service, it must also be named and registered in the RMI Registry).

The **Java 2 SDK** implementation of RMI adds a new semantic for the client-server connection. In this version, RMI supports activatable remote objects. When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant. If it is dormant, RMI will instantiate the object and restore its state from a disk file. Once an activatable object is in memory, it behaves just like JDK 1.1 remote service implementation objects.

17.4.4 Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP. Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack.

TCP/IP provides a persistent, stream-based connection between two machines based on an IP address and port number at each end. Usually a DNS name is used instead of an IP address. In the current release of RMI, TCP/IP connections are used as the foundation for all machine-to-machine connections. On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP).

Note: JRMP is a proprietary, stream-based protocol that is only partially specified. It is now available in two versions. The first version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server. The second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes.

17.4. 5 Bootstrapping and the RMI registry

For a client and a server to start talking, they need some way to connect. Acquiring this connection is known as *bootstrapping*. RMI provides the *RMI registry* for this purpose. When an object has to be remotely accessible, it has to be first registered with the registry, in a particular name. After this, the registry will route all incoming requests for that name to the object. The RMI registry can be compared to a giant hashtable that maps names to objects.

The RMI registry accomplishes this task by sitting at a well-known network port and listening for incoming connections. When the registry receives a request, it looks up the name of the remote object requested, and returns the stub for that remote object to the client. You can have many RMI registries on a machine but each one must be on its own port. RMI uses port 1099 by default. The RMI registry can be started in two ways:

- By typing rmiregistry from the command line, or
- From inside a Java program using the java.rmi.Registry class.

Looking up a remote object

The lookup() method of the java.rmi.Naming class takes an RMI URL, connects to an RMI registry on a target JVM, and returns a java.lang.Object representing the remote object. The returned object is actually the remote stub, and it can be cast to the remote object's interface type.

RMIC - The RMI compiler

The rmic tool supplied by Sun MicroSystems is used to generate a stub and a skeleton from a remote object implementation. This is called on the class file in a similar manner to the way the Java interpreter itself is invoked.

17.5 Working of RMI

To understand how RMI works, you must appreciate the roles of stubs and skeletons as depicted in the figure-3.3 Here, an object C on machine 1 wants to call a method provided by Object S on machine 2. Object S is a remote object. In other words, it implements one or more remote interfaces that contain methods, which can be invoked by Java objects on other machines.
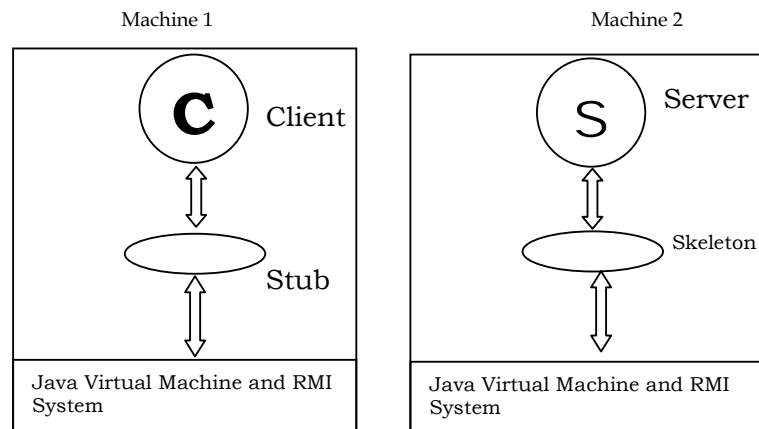

Figure 3.3 RMI Work

However, because object S is in a different address space on a separate machine, object C cannot simply invoke such a method directly. Instead, a stub is provided that executes in the same address space as object C. This acts as a proxy for the remote object S. The stub has the following primary responsibilities:

- First, it presents the same interfaces as object S. Therefore, from the perspective of object C the stub is equivalent to the remote object.

- Second, the stub works with the JVM and RMI system on machine 1 to serialize any arguments to a remote method call and send this information to machine 2.

- Finally, the stub receives any result from the remote method invocation and returns this to object C.

Following are the primary responsibilities of the skeleton:

- First, it receives the remote method call and any associated arguments. It works with the JVM and RMI system on machine 2 to deserialize any arguments for this remote method call.

- Second, it invokes the appropriate method of object S with these arguments.

- Finally, the skeleton receives any return value from this method call and works with the JVM and RMI system on machine 2 to serialize this return value and send this information back to machine 1.

The object serialization facilities are used to send objects from one machine to another. Objects can be supplied as arguments to a remote method call or returned as results from it.

Object C gets a reference to the correct stub for object S registered in the RMI registry through the URL that uses the RMI or TCP/IP protocol. The protocol is of the form:

rmi://host:port/server

where host is the IP address or name of the server machine on which object S resides, port is the optional port number of the registry on that machine, and server is the name of the remote server.

Therefore, if object C wants to contact object S on the machine with IP address a.b.c.d, it can obtain a stub for that remote object by asking the Registry for a reference to the URL shown below:

rmi://a.b.c.d/DivideServer

Once object C has a reference to the Stub for object S, it can issue method calls to the stub.
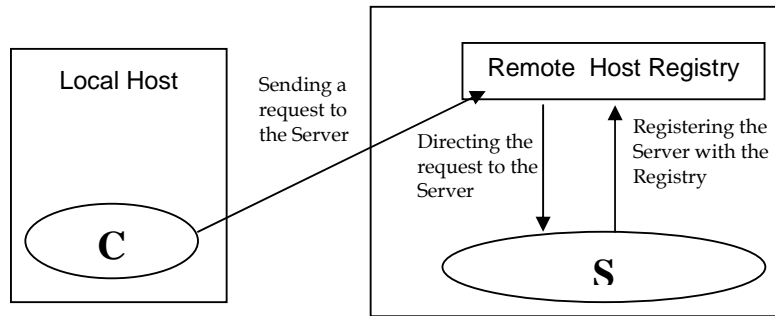

Figure 3.4: Remote objects register themselves

The Java garbage collection facilities also work in a distributed application. Remote objects are Garbage collected automatically, just as local objects are. However, the current distributed collector uses reference counting and cannot detect cycles of unreferenced objects. Cycles must be explicitly broken by the programmer.

17.6 advantages of RMI

The following are the advantages of RMI:

- **Object Oriented:** RMI can pass full objects as arguments and return values, not just predefined data types. This means that you can pass complex types, such as a standard Java hashtable object, as a single argument. In existing RPC systems you would have to have the client decompose such an object into primitive data types, ship those data types, and then recreate it on the server.

- **Mobile Behavior:** RMI can move a behavior (class implementations) from client to server and server to client. For example, you can define an interface for examining employee expense reports to see whether they conform to current company policy. When an expense report is created, an object that implements that the client can fetch interface from the server. When the policies change, the server will start returning a different implementation of that interface that uses the new policies. The constraints will therefore be checked on the client side-providing faster feedback to the user and less load on the

server-without installing any new software on user's system. This gives you maximal flexibility, since changing policies requires you to write only one new Java class and install it once on the server host.

- **Design Patterns:** Passing objects lets you use the full power of object oriented technology in distributed computing, such as two- and three – tier systems. When you can pass behavior, you can use object oriented design patterns in your solutions. All object oriented design patterns rely upon different behaviors for their power.

- **Safe and Secure:** RMI uses built-in Java security mechanisms that allow your system to be safe when users download implementations. RMI uses the security manager defined to protect systems from hostile applets to protect your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.

- **Easy to Write/Easy to Use:** RMI makes it simple to write remote Java servers and Java clients that access those servers. A remote interface is an actual Java interface. A server has roughly three lines of code to declare itself a server, and otherwise is like any other Java object. This simplicity makes it easy to write servers for full-scale distributed object systems quickly, and to rapidly bring up prototypes and early versions of software for testing and evaluation. Since RMI programs are easy to write they are also easy to maintain.

- **Connects to Existing/Legacy Systems:** RMI interacts with existing systems through Java's native method interface JNI. RMI and JNI enable us to write the client in Java and use an existing server implementation. Using RMI/JNI to connect to existing servers, we can rewrite any part of the server in Java, and get the full benefits of Java in the new code. Similarly, RMI interacts with existing relational databases using JDBC without modifying existing non-Java source that uses the databases.

- **Write Once, Run Anywhere:** RMI is part of Java's "Write Once, Run Anywhere" approach. Any RMI based system is 100% portable to any Java Virtual Machine, as is an RMI/JDBC system. If you use RMI/JNI to interact with an existing system, the code written using JNI will compile and run with any Java virtual machine.

- **Distributed Garbage Collection:** RMI uses its distributed garbage collection feature to collect remote server objects that are no longer referenced by any clients in the network. Analogous to garbage collection inside a Java Virtual Machine, distributed garbage collection lets you define server objects as needed, knowing that they will be removed when they are no longer needed by clients.

- **Parallel Computing:** RMI is multi-threaded, allowing the servers to use Java threads for better concurrent processing of client requests.

- **The Java Distributed Computing Solution:** RMI is part of the core Java platform starting with JDK 1.1. So, it exists on every 1.1 Java Virtual Machine. All RMI systems use the same public protocol, so all Java systems can talk to each other directly, without any protocol translation overhead.

17.7 Short Summary

- Distributed object applications need to:

  - ❖ Locate remote objects
  - ❖ Communicate with remote objects
  - ❖ Load class bytecodes for objects that are passed as parameters or return values

- The Java distributed object model differs from the Java object model …
- The rmic tool supplied by Sun MicroSystems is used to generate a stub and a skeleton from a remote object implementation

- The skeleton is a helper class for RMI

- The Internet and the Web are examples of distributed systems that have been developed using the TCP/IP Client/Server methodology.

17.8 Brain Storm

1. Write a note on Distributed application?

2. What are the advantages of RMI?

ಬಿಜ

Lecture 18

Client/Server Application

Objectives

In this Lecture you will learn how to .....

✎   Create the Remote Interface

✎   Create the class that implements the Remote Interface

✎   Create the main server program.

✎   Create Stub and Skeleton Classes.

✎   Copy the Remote Interface and Stub File to the Client Host.

✎   Create a Client class that uses the remote services.

✎   Start up the Registry, Server and Client.

Coverage Plan

Lecture 18

18.1 Snap Shot

A Simple Client/Server Application

This Section describes how to build a simple client/server application using RMI. The Server receives a request from client, processes it, and returns a result. In this example, the request includes two double numbers. The server divides these and returns the result.

18.2 Create the Remote Interface

Remote method invocations can fail in different ways from local Method invocations, due to network related communication problems and server problems. To indicate that it is a remote object, an object implements a remote Interface.

Remote Objects are referenced through interfaces. In order to implement a remote object, we must first create an interface for that object. This interface must be public and must extend the java.rmi.Remote interface. The remote methods that have to be accessible to the client should be defined within this interface. Each of these methods must declare a RemoteException in its throws clause in addition to any application-specific exceptions. Also, a remote object passed as an argument or return value (either directly or embedded within a local object) must be declared in the remote interface and not the implementation class. The file DivideServer.java given below defines the remote interface that is provided by the server. It contains one method that accepts two double arguments and returns the value computed by dividing the first argument by the second argument.

```
package divide;
import java.rmi.*;

public interface DivideServer extends Remote {
      double divide(double d1, double d2) throws RemoteException;
}
```

- Save the code as DivideServer.java and compile it.

Note: The Remote Interface in the java.rmi package defines no constants or methods .It exists only to designate which interfaces are remote. Every remote interface must directly or indirectly extend java.rmi.Remote. Only remote interfaces can be invoked through RMI. Local interfaces can not be called in this manner.

18.2.1 Create a class that implements the Remote Interface

The file DivideServerImpl.java implements the remote interface. It should also implement all the methods defined in the remote Interface. All remote objects typically extend from the UnicastRemoteObject class. However, it can also extend from other subclasses of the RemoteServer class .The UnicastRemoteObject class provides the functionality needed to make objects available from remote machines. Extending UnicastRemoteObject indicates that the DivideServerImpl class is used to create a single non-replicated remote object that uses the RMI's default sockets-based transport for communication.

```
package divide;
import java.rmi.*;
import java.rmi.server.*;

public class DivideServerImpl extends
UnicastRemoteObject
implements DivideServer {
public DivideServerImpl()throws RemoteException {
}
public double divide(double d1, double d2)throws
RemoteException {
return d1/d2;
}
}
```

- Save the code as DivideServerImpl.java and compile it

Note:
- The RemoteObject class in the java.rmi.server package extends java.lang.Object and overrides the equals(), hashcode() and toString() methods to provide the correct behavior for remote objects.
- The RemoteServer class in the java.rmi.server package extends RemoteObject. It is an abstract class that defines the methods needed to create and export remote objects.
- The UnicastRemoteObject class in the java.rmi.server package is a concrete subclass of RemoteServer. This class defines a non-replicated remote object whose references are valid only while the server process is alive. This class provides support for point-to-point active object references (invocations, parameters and results) using TCP streams.

18.2.2 Create the main Server program

The file DivideServerApp.java contains the main program for the server machine. The main() method uses the setSecurityManager() method of the System class to set an object to be used as the remote object's security manager. A security manager needs to be running so that it can guarantee that the classes loaded do not perform sensitive operations. If no security manger is specified, no class loading for RMI classes is allowed. The Remote object should register a name by which it can be remotely referenced, with the RMI registry. This is done by using the rebind() method of the Naming class. This method associates a name with an object reference. The first argument to this method is a string that names the server, here it is "DivideServer". Its second argument is an instance of the server, here it is DivideServerImpl. The following listing gives the program. It basically informs the user that it has successfully completed the registration.

```
package divide;
import java.net.*;
import java.rmi.*;

public class DivideServerApp  {
public static void main(String args[])  {

 try {
   DivideServerImpl divideServerImpl;
   divideServerImpl = new DivideServerImpl();
   Naming.rebind("DivideServer",divideServerImpl);
  }
 catch (Exception ex){
      ex.printStackTrace();
  }
 }
}
```

Save the file as DivideServerApp.java and compile it

Note: Installing a Security Manager is not absolutely necessary. It is useful only when we want to constrain the actions in the server implementation or if the server is itself an RMI client of another server.

The Naming class in the java.rmi package provides three methods to associate names with remote objects. These are :

```
static void bind(String name,Remote robj) throws AlreadyBoundException,
                                                MalformedURLException
static void rebind(String name,Remote robj) throws RemoteException
static void unbind(String name,Remote robj) throws
NotBoundException,RemoteException, MalformedURLException
```

where name is the name of the object and robj is a reference to that object.

The bind() method associates a name with the remote object. However, if the name is already used in the registry, an AlreadyBoundException is thrown. The rebind() method also associates a name with the remote object. If the name is already used in the registry, the existing binding is replaced. Finally, the unbind() method removes the binding for the name.

18.2.3 Create Stub and Skeleton Classes

To generate stubs and skeletons, you use a tool called the RMI compiler. To generate the stub and skeleton for DivideServerImpl, give the following command at the command prompt:

```
rmic –d . divide.DivideServerImpl
```

The –d option specifies the path of the class. In this case, it is the current directory. This command generates files named DivideServerImpl_Skel.class and DivideServerImpl_Stub.class in the subdirectory divide.

Note: rmic is dealt in detail in the **Appendix** section at the end of this chapter.

18.2.4 Copy the Remote Interface and Stub File to the Client Host

Copy the interface file (**DivideServer.java**) and the stub file (**DivideServerImpl_stub.java)** to an appropraite location in the client machine.

18.2.5 Create a Client class that uses the remote services

The file DivideClient.java implements the client side of this distributed application. It requires three command line arguments. The first is the IP address or name of the server machine. The second and third arguments are the numbers to be divided.

The application begins by forming a string that follows the URL syntax. This URL uses the RMI protocol. It includes the IP address or name of the server and the string "DivideServer". It then invokes the lookup() method of the Naming Class. This accepts one argument that is the rmi URL and returns a reference to an object type Remote which is then cast to a DivideServer. All remote method invocations can then be directed to this object. The syntax for the lookup() method is: static Remote lookup(String url) throws NotBoundException, RemoteException, MalformedURLException,UnKnownHostException

The Naming class also contains the list() method that returns an array of strings of the URLs in the registry located at the given URL.

static String[] list(String url)

In order to set up a security manager for the client application, we give the following command:

System.setSecurityManager(new RMISecurityManager());

This is to ensure that the classes loaded do not perform "Sensitive" operations. In the above case RMI generates security errors when you attempt to run the client program on some systems. If you get AccessControlException error messages associated with calls to the Naming.lookup() method, your system needs to be configured so that RMI calls can execute successfully.

One way to do this is to set up a policy, which grants all the permissions.
grant {
        permission java.security.AllPermission ;
}

Enter the above code into the file c:\jdk1.2\jre\lib\security\java.policy.
Save it, and run your client program.

> **Note**: Here c:\jdk1.2\jre is the <java.home > Directory, i.e, the directory in which Java Runtime Environment is installed. Check out for the home directory in your system.

The program continues by displaying its arguments and then invokes the remote divide() method. The result returned from this method is displayed.

```java
package divide;
import java.rmi.*;
public class DivideClient {
public static void main(String args[]) {

  try {
   // make the rmi URL to name DivideServer
   System.setSecurityManager(new RMISecurityManager());
   String divideServerURL;
   divideServerURL="rmi://" + args[0] +
       "/DivideServer";
   // Obtain the reference to that remote object
       DivideServer divideServer;
   divideServer=(DivideServer)Naming.lookup(divideServerURL);
   //Display numbers
   System.out.println("The first number is: " +      args[1]);
   double d1=Double.valueOf(args[1]).doubleValue();
   System.out.println("The second number is: " +      args[2]);
   double d2=Double.valueOf(args[2]).doubleValue();

   // Invoke remote method and display result
   double result = divideServer.divide(d1,d2);
   System.out.println("The result is: "+ result);
  } catch(Exception ex)  {
    ex.printStackTrace();
   }
 }
}
```
- Compile the file DivideClient.java

18.2.6 Start up the Registry, Server and Client

Start the RMI Bootstrap Registry

The RMI Registry is a simple server-side bootstrap name server that allows remote client to get reference to a remote object.

To start the registry on the server, execute the rmiregistry command. This command produces no output and is typically run in the background. This program listens on the default port no. 1099 for incoming requests to access named objects. At the command prompt, type: start rmiregstry

---

Note: More about RMI Registry is given in **Appendix** section at the end of this chapter.

Start the Server

Start the Server code from the command prompt by giving the following command: java divide.DivideServerApp

Start the Client

The DivideClient.java program requires three arguments: the name or IP address of the server machine and the two numbers to be divided. You may invoke it from the command line by using one of the two formats shown below:

java divide.DivideClient rad-wm-46 8 2

java divide.DivideClient 127.0.0.1 8 2

In the first line, the name of the server is provided. The second line uses its IP address (e.g. 127.0.0.1).

In either case, sample output form this program is shown here:

The first number is : 8
The second number is : 2
The result is : 4. 0

Note that if you divide 0 by 0 the result is "NaN" which means "Not a Number." If you divide 1 by 0 the result is "Infinity."

Building an RMI Client Applet

Now let's make our client as an applet.

```java
package divide;
import java.rmi.*;
import java.awt.event.*;
import java.awt.*;
import java.applet.*;

public class DivideClient extends java.applet.Applet {

    DivideServer divideServer;
    public void init () {
      try {
        // make the rmi URL to name DivideServer
        System.setSecurityManager(new RMISecurityManager());
        String divideServerURL;
      divideServerURL="//" + getParameter("url") +   "/DivideServer";

        // Obtain the reference to that remote object
        divideServer =    (DivideServer)Naming.lookup(divideServerURL);

      }catch(Exception ex) {
          ex.printStackTrace();
       }
    }
    public void paint(Graphics g){
      try {
        //Display numbers
 g.drawString("The first number is: " +        getParameter("first"),40,40);
        double d1=Double.valueOf(getParameter("first")).doubleValue();
g.drawString("The second number is:"+
getParameter("second"),40,60);
        double d2=Double.valueOf(getParameter("second")).doubleValue();

        // Invoke remote method and display result
  double result = divideServer.divide(d1,d2);
         g.drawString("The Result is : " +
result,40,80);

      } catch(Exception e){
          }
    }
```

```
}
Save the above applet as DivideClient.java and compile it

Create the following Html file for the Applet:

<html>
 <head>
  <title>
   test applet
  </title>
 </head>
 <body bgcolor = cyan>
<applet code = DivideClient height =240 width = 400>
    <param name = url value = master2 >
    <param name = first value = 8 >
    <param name = second  value = 2 >
   </applet>
  </body>
</html>
```
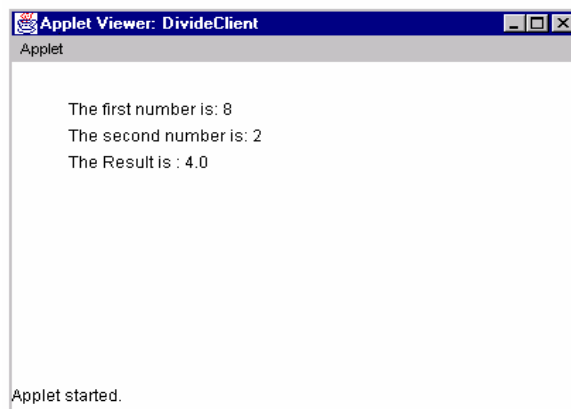
- Save the above HTML code as DivideHTML.html

  To run the applet, give the following command at the command
  prompt:

  appletviewer DivideHTML.html

The output of the program appears as shown in figure 3.5.

Figure 3.5 The out put of the program

How RMI simulates pass by reference

Passing parameters by value can lead to inefficiencies if the referenced graph of objects is very large. For such cases, RMI simulates a pass-by-reference convention. If you want a parameter by reference, the parameter itself must be a remote object. What is actually passed is the remote object's stub. Stubs are network-aware references to remote objects. java.rmi.RemoteStub objects are the manifestation of those remote references to objects. These are the exact same stubs described earlier that are generated by rmic. Because Java RMI stubs are also serializable, they are passable over the network. This is what saying that all parameters are passed by value means and that pass-by-reference is simulated.

18.3 Short Summary

- A security manager needs to be running so that it can guarantee that the classes loaded do not perform sensitive operations. If no security manger is specified, no class loading for RMI classes is allowed.

- The RMI Registry is a simple server-side bootstrap name server that allows remote client to get reference to a remote object.

18.4 Brain Storm

1. Write about creation of a simple client/server application using RMI.

৮০৫৪

Lecture 19

Dynamic Class Loading

Objectives

In this lecture you will learn about....

✎   Codebase in applets

✎   Codebase in RMI

✎   Command-line examples

✎   An Example of Dynamic Class Loading

Coverage Plan

Lecture 19

19.1 Snap Shot

One of the most significant capabilities of the JavaTM platform is the ability to dynamically download Java software from any Uniform Resource Locator (URL) to a Java Virtual Machine (JVM) running in a separate process, usually on a different physical system. The result is that a remote system can run a program, for example an applet, which has never been installed on its disk.

For example, a JVM running from within a web browser can download the bytecodes for the subclasses of java.applet.Applet and any other classes needed by that applet. The system on which the browser is running has most likely never run this applet before, nor installed it on its disk. Once all the necessary classes have been downloaded from the server, the browser can start the execution of the applet program using the local resources of the system on which the client browser is running.

Java RMI takes advantage of this capability to download and execute classes on systems where those classes have never been installed on disk. Using the RMI API any JVM, not only those in browsers, can download any Java class file including specialized RMI stub classes, which enable the execution of method calls on a remote server using the server system's resources.

The notion of codebase originates from the use of ClassLoaders in the Java programming language. When a Java program uses a ClassLoader, that class loader needs to know the location(s) from which it should be allowed to load classes. Usually, a class loader is used in conjunction with an HTTP server that is serving up compiled classes for the Java platform.

Codebase:

A codebase can be defined as a source or a place from which to load classes into a Java virtual machine. For example, if you invited a new person to your house, you would need to give that person the directions to the place where you live, so that he or she could locate your house. Similarly, you can think of a codebase as the directions that you give to a JVM, so it can find your [potentially remote] classes.

You can think of your CLASSPATH as a "local codebase", because it is the list of places on disk from which you load local classes. When loading classes from a local disk-based source, your CLASSPATH variable is consulted. Your CLASSPATH can be set to take either relative or absolute path names to directories and/or archives of class files. So

---

just as CLASSPATH is a kind of "local codebase", the codebase used by applets and remote objects can be thought of as a "remote codebase".

19.2 Codebase in applets

To interact with an applet, that applet and any classes that it needs to run must be accessible by remote clients. While applets can be accessed from "ftp://" or local "file:///" URLs, they are usually accessed from a remote HTTP server.

1. The client browser requests an applet class that is not found in the client's CLASSPATH
2. The class definition of the applet (and any other class(es) that it needs) is downloaded from the server to the client using HTTP
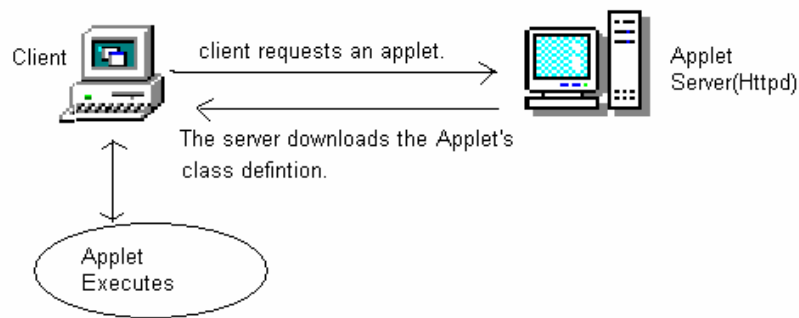3. The applet executes on the client

Figure 3.6 Downloading

The applet's codebase is always relative to the URL of the HTML page in which the <applet> tag is contained.

19.3 Codebase in RMI

Using RMI, applications can create remote objects that accept method calls from clients in other JVMs. In order for a client to call methods on a remote object, the client must have a way to communicate with the remote object. Rather than having to program the client to speak the remote object's protocol, RMI uses special classes called stubs that can be downloaded to the client that are used to communicate with (make method calls on) the remote object.

The java.rmi.server.codebase property value represents one or more URL locations from which these stubs (and any classes needed by the stubs) can be downloaded. Like applets, the classes needed to execute remote method calls can be downloaded from "file:///" URLs, but like applets, a "file:///" URL generally requires that the client and the server reside on the same physical host, unless the file system referred to by the URL is made available using some other protocol, such as NFS. Generally, the classes needed to execute remote method calls should be made accessible from a network resource, such as an HTTP or FTP server.
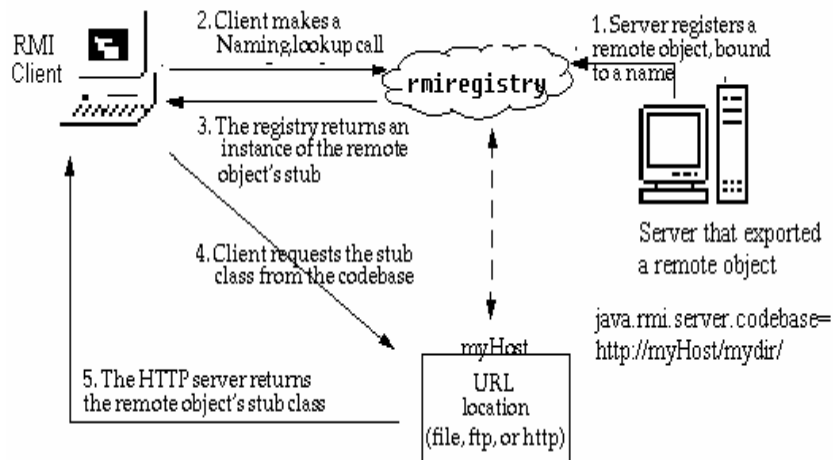
The remote object's codebase is specified by the remote object's server by setting the java.rmi.server.codebase property. The RMI server registers a remote object, bound to a name, with the RMI registry. The codebase set on the server JVM is annotated to the remote object reference in the RMI registry.

1. The RMI client requests a reference to a named remote object. The reference (the remote object's stub instance) is what the client will use to make remote method calls to the remote object.

2. The RMI registry returns a reference (the stub instance) to the requested class. If the class definition for the stub instance can be found locally in the client's CLASSPATH, which is always searched before the codebase, the client will load the class locally. However, if the definition for the stub is not found in the client's CLASSPATH, the client will attempt to retrieve the class definition from the remote objects codebase.

3. The client requests the class definition from the codebase. The codebase the client uses is the URL that was annotated to the stub instance when the stub class was loaded by the

registry. Back in step 1, the annotated stub for the exported object was then registered with the RMI registry bound to a name.

4. The class definition for the stub (and any other class (es) that it needs) is downloaded to the client.

Note: Steps 4 and 5 are the same steps that the registry took to load the remote object class, when the remote object was bound to a name in (registered with) the RMI registry. When the registry attempted to load the remote object's stub class, it requested the class definition from the codebase associated with that remote object.

---

1. Now the client has all the information that it needs to invoke remote methods on the remote object. The stub instance acts as a proxy to the remote object that exists on the server; so unlike the applet which uses a codebase to execute code in its local JVM, the RMI client uses the remote object's codebase to execute code in another, potentially remote JVM, as Illustrated in Fig 3.8
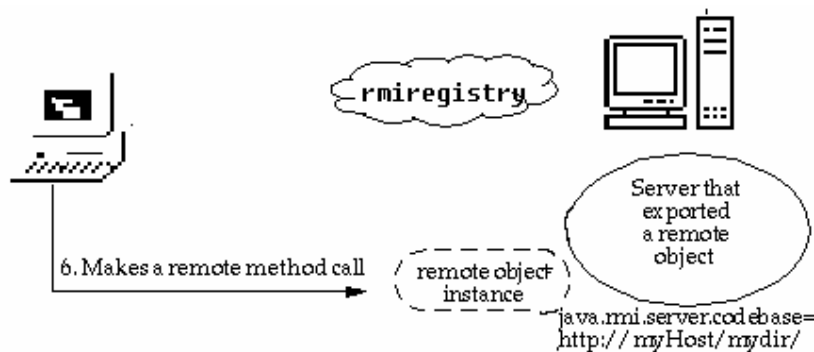

Figure 3.8: RMI client making a remote method call

19.4 Command-line examples

In the case of an applet, the applet codebase value is embedded in an HTML page. In the case of Java RMI codebase, rather than having a reference to the class embedded in an HTML page, the client first contacts the RMI registry for a reference to the remote object. Since the remote object's codebase can refer to any URL, the value of the RMI codebase must be an absolute URL to the location of the stub class and any other classes needed by the stub class. This value of the codebase property can refer to:

*   The URL of a directory in which the classes are organized in package-named sub-directories
*   The URL of a JAR file in which the classes are organized in package-named directories

---

- A space-delimited string containing multiple instances of JAR files and/or directories that meet the criteria above

Examples

If the location of your downloadable classes is on an HTTP server named "masdemo", in the directory "mydir" (under the web root), your codebase property setting might look like this:

- Djava.rmi.server.codebase=http://masdemo/mydir/

If the location of your downloadable classes is on an HTTP server named "master2", in a JAR file named "myfile.jar", in the directory "public" (under the web root), your codebase property setting might look like this:

- Djava.rmi.server.codebase=http://master2/public/myFile.jar

Now let's suppose that the location of your downloadable classes has been split between two JAR files, "myFile.jar" and "myOtherFile.jar". If these JAR files are located on different servers (named "master1" and "master2"), your codebase property setting might look like this:

```
-Djava.rmi.server.codebase="http://master1/myFile.jar
   http://master2/myOtherFile.jar"
```

19.5 An Example of Dynamic Class Loading

Simple.java: Interface defining the remote behavior

```
public interface Simple extends java.rmi.Remote {
      String call(String message) throws java.rmi.RemoteException;
}
```

SimpleServer.java: Waits for client connections and returns simple message back to the client

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
```

```
public class SimpleServer
extends UnicastRemoteObject implements Simple{
     public SimpleServer() throws RemoteException {
          super();
     }

     public String call(String message) throws RemoteException    {
return "From SimpleServer: Thanks for your message: "
 +message;
     }

     public static void main(String args[])  {
         // Create and install the security manager
         System.setSecurityManager(new RMISecurityManager());
         try {
  // create an SimpleServer server and register it
              SimpleServer obj = new SimpleServer();
              Naming.rebind(url,obj);
              System.out.println("Echo Server ready.");

} catch (Exception e) {
System.out.println("SimpleServer.main: an exception occurred: " +
e.getMessage());
e.printStackTrace();
  }
     }
}
```

SimpleClient.java: Echo client program, which invokes the remote method

```
import java.rmi.*;
import java.rmi.server.*;
public class SimpleClient {

   public static void main(String args[])
   {
       // Create and install the security manager
       System.setSecurityManager(new RMISecurityManager());
       try {
           System.out.println("SimpleClient:");
           String url = "rmi://" + args[0] + "/Simple";

           // lookup Simple server
```

```
    System.out.println("Trying to lookup remote object:");
    SimpleInterafce obj = (SimpleInterface) java.rmi.Naming.lookup(url);

        // call remote method
        String message = obj.call("Hi SimpleServer");

        // print message returned from server
        System.out.print("Message from Simple Server: ");
        System.out.println("\t" + message + "\n");

     } catch (Exception e) {
         System.out.println("SimpleClient: an exception   occurred: " +
         e.getMessage());
         e.printStackTrace();
         }
   }
 }
```

Compile the Client program using the command:

> javac SimpleClient.java

Compile the Remote Interface using the command:

> javac SimpleInterface.java

Compile the Server program using the command:

> javac SimpleServer.java

Create the stub and skeleton using the command:

> rmic SimpleServer

Start the rmiregistry using the command:

> rmiregistry &

Run the Server using the command:

> java SimpleServer
>
> -Djava.rmi.server.codebase=http://*hostname:port/location/ hostname*

Run the Client as follows:

To run the client code, you have to specify the host name of the server at the command prompt. The port number is required if it has been used for the rmiregistry. Client can reside in a different machine from the Server.

java SimpleClient *hostname*

19.6 Short summary

- When the codebase property value is set to the URL of a directory, the value must be terminated by a "/".

- Specification of the host name of the server at the command prompt is need to run the client code.

- A codebase can be defined as a source or a place from which to load classes into a Java virtual machine

- One of the most significant capabilities of the JavaTM platform is the ability to dynamically download Java software from any Uniform Resource Locator…

19.7 Brain Storm

1. What is a Codebase?

2. Write a short note on codebase in
    a. Applet
    b. RMI.

ॐ

Lecture 20

Trouble Shooting Tips

Objectives

In this Lecture you are going to learn about

- ❧ Trouble shooting tips

- ❧ common problems associated with the java.rmi.server.codebase property

```
┌─────────────────────────────────────────────────┐
│ ┌─────────────────────────────────────────────┐ │
│ │              Coverage Plan                    │ │
│ └─────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────┘
```

Lecture 20

20.1 Snap Shot

This chapter discuss on troubleshooting tips and the common problems associated with the java.rmi.server.codebase property, which are discussed below:

20.2 Troubleshooting Tips

Any serializable class, including RMI stubs, can be downloaded if your RMI programs are configured properly. Here are the conditions under which dynamic stub downloading will work:

a. The stub class and any of the classes that the stub relies on are served up from a URL reachable from the client.

b. The java.rmi.server.codebase property has been set on the server program (or in the case of activation, the "setup" program) that makes the call to bind() or rebind() methods, such that:

- The value of the codebase property is the URL in step A and
- If the URL specified as the value of the codebase property is a directory, it must end in a trailing "/"

a. The rmiregistry cannot find the stub class or any of the classes that the stub relies on in its CLASSPATH. This is so the codebase gets annotated to the stub when the registry does its class load of the stub, as a result of calls to bind or rebind in the server or setup code.

b. The client has installed a SecurityManager that allows the stub to be downloaded. In the Java 2 SDK, Standard Edition, v 1.2 this means that the client must also have a properly configured security policy file.

There are two common problems associated with the java.rmi.server.codebase property, which are discussed below:

20.2.1 Problem while running the RMI server

The first problem you might encounter is the receipt of a ClassNotFoundException when attempting to bind or rebind a remote object to a name in the registry. This exception is usually due to a malformed codebase property, resulting in the registry not being able to locate the remote objects stubs or other classes needed by the stub.

It is important to note that the remote object's stub implements all the same interfaces as the remote object itself, so those interfaces, as well as any other custom classes declared as method parameters or return values, must also be available for download from the specified codebase.

Most frequently, this exception is thrown as a result of omitting the trailing slash from the URL value of the property. Other reasons would include: the value of the property is not a URL; the path to the classes specified in the URL is incorrect or misspelled; the stub class or any other necessary classes are not all available from the specified URL.

The exception that you may encounter in such a case would look like this:

```
java.rmi.ServerException: RemoteException occurred in server thread; nested
exception is:
      java.rmi.UnmarshalException: error unmarshalling arguments; nested
exception is:
      java.lang.ClassNotFoundException:
examples.callback.SimpleSerever_Stub
java.rmi.UnmarshalException: error unmarshalling arguments; nested exception
is:
      java.lang.ClassNotFoundException:
examples.callback.SimpleServer_Stub
java.lang.ClassNotFoundException:
examples.callback.SimpleServer_stub
      at
sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(Compiled
Code)
      at sun.rmi.transport.StreamRemoteCall.executeCall(Compiled Code)
      at sun.rmi.server.UnicastRef.invoke(Compiled Code)
      at sun.rmi.registry.RegistryImpl_Stub.rebind(Compiled Code)
      at java.rmi.Naming.rebind(Compiled Code)
      at examples.callback.SimpleServer.main(Compiled Code)
RemoteException occurred in server thread; nested exception is:
      java.rmi.UnmarshalException: error unmarshalling arguments; nested
exception is:
      java.lang.ClassNotFoundException:
examples.callback.MessageReceiverImpl_Stub
```

```
20.2.2 Problem while running the RMI client
```

```
 java.rmi.UnmarshalException: Return value class not found; nested exception
is:
```

```
      java.lang.ClassNotFoundException: MyImpl_Stub
      at
sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:109
      at java.rmi.Naming.lookup(Naming.java:60)
      at RmiClient.main(MyClient.java:28)
```

20.3 Object Activation

In the previous examples of RMI, in order to obtain a reference to a remote object, the server that generated the instance of the object had to be running in a JVM. This would sufficient for most of the applications where there is one Server object and one client. However, for large systems that create a number of objects those are not used at the same time, it is useful to suspend those objects until they are needed. This in turn gives the facility for the application developer for not running the server always. Only when the client requests for an object reference the server is activated. This is accomplished by an object daemon and the whole mechanism is called activation mechanism.

Activation allows a Java object to be bound (named by the Registry) and then "activated" at some later date simply by referencing the object through the Registry. The primary benefit of this approach is that the application that creates the instance of the remote object can terminate or exit normally before the object is ever used. The ability to activate remote objects on requests allows RMI system designers much greater flexibility in designing smaller servers. The main mechanism to make activation work is another daemon process, the Java RMI Activation System Daemon (rmid).
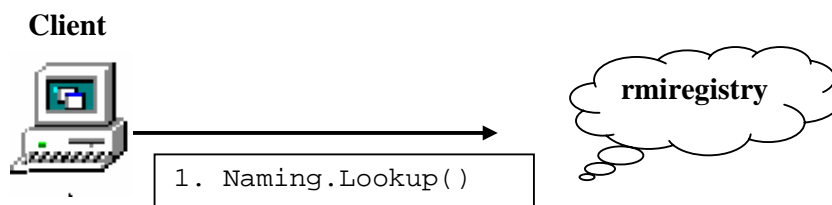

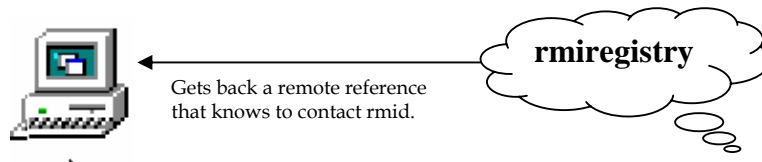**Figure 20.1** : A Client asking a reference of a remote object

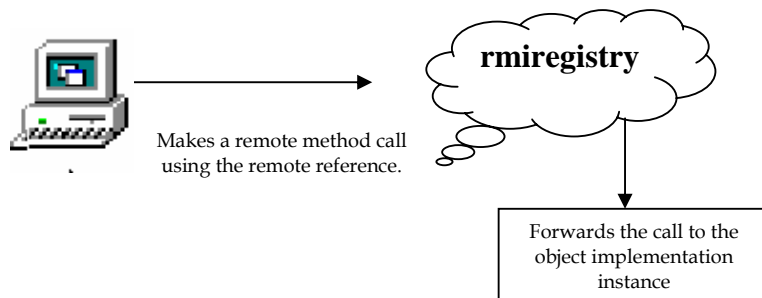**Figure 20.2** : A Client asking a reference of a remote object.

Figure 20.3 A Client making a remote call that is directed to the rmid.

For the Activation to work the rmid (daemon process) should be running .To run the daemon process type the following at command prompt,

C:\> start rmid

Note: By, default rmid will start on port number 1098, but an alternate port may be specified. C:\> start rmid -port 2002

- *Transient remote object reference* - is a reference to an instance of a remote object. When a Remote object is in no longer active, the live reference is no longer valid.

- *Persistent object activation ID* – is an object identifier that is valid whether the object is active or inactive. It provides a reference to the object's activator.

When a client object invokes a method of an Activatable remote object, the transient remote object reference is checked to determine whether it is valid. If it is not valid, the remote object's activation ID is used to activate the remote object is performed using the *activation protocol*. It makes use of a special object, known as Activator. It is a special object executed on

a remote host as the result of running the rmid process. It maintains a pool of information that maps activatable objects to information that is needed to activate them, such as an objects class name and its code source location.

Each executing JVM is associated with an activation group. The activation group of a JVM is used to activate objects on that JVM. When the Activator receives a faulting remote reference, it uses the object's activation ID (included in the reference) to look up the object's activation descriptor. The activation descriptor tells the Activator what class to load, where to load it from (its source code), where to activate it (its activation group), and how to initialize the object. The Activator checks for an executing JVM with the required group. If none is found, it creates one. The Activator then requests the activation group to activate the object with in the JVM. The activation group returns a valid live reference to the activated object, which is passed, back to the client object making the method invocation.

When a client object invokes a method of an Activatable remote object, the transient remote object reference is checked to determine whether it is valid. If it is not valid, the remote object's activation ID is used to activate the remote object is performed using the *activation protocol*. It makes use of a special object, known as Activator. It is a special object executed on a remote host as the result of running the rmid process. It maintains a pool of information that maps activatable objects to information that is needed to activate them, such as an objects class name and its code source location.

Each executing JVM is associated with an activation group. The activation group of a JVM is used to activate objects on that JVM. When the Activator receives a faulting remote reference, it uses the object's activation ID (included in the reference) to look up the object's activation descriptor. The activation descriptor tells the Activator what class to load, where to load it from (its source code), where to activate it (its activation group), and how to initialize the object. The Activator checks for an executing JVM with the required group. If none is found, it creates one. The Activator then requests the activation group to activate the object with in the JVM. The activation group returns a valid live reference to the activated object, which is passed, back to the client object making the method invocation.

20.4 Short summary

- A remote object is active when it is instantiated within a Java Virtual Machine and exported for access via RMI.

- A remote object is inactive when it is not yet instantiated or exported

- Activation is the process of transforming in Inactive object into an Active object.

- RMI uses a form of remote activation that is referred to as lazy Activation. A remote object is not activated until one of its methods has been invoked.

- Lazy Activation is implemented using a technique called faulting remote reference…

20.5 Brain Storm

1. Explain the common problems associated with the java.rmi.server. codebase property.

2. What are the conditions under which dynamic stub downloading will work?

ఞఞ

Lecture 21

Making an Object Activatable

Objectives

In this lecture you are going to learn

✘   About making an object Activatable

Coverage Plan

Lecture 21

21.1 Snap Shot

In this chapter you are going to learn about making an object to activatable.

21.2 The Remote Interface

```
import java.rmi.*;
public interface ActivationServer extends Remote
{
      double divide(double d1, double d2) throws RemoteException;
}
```

21.3 The Implementation Class

Step 1: Make the appropriate imports in the implementation class

```
import java.rmi.*;
import java.rmi.activation.*;
```

Step 2: Extend your class from java.rmi.activation.Activatable

```
public class ActivationServerImpl extends Activatable
    implements ActivationServer {
```

Step 3: Declare a two-argument constructor in the implementation class

The following statements register the object with the activation system and then export it to a port

```
public ActivationServer(ActivationID id, MarshalledObject data)
    throws RemoteException {
            super(id, 0);
}
```

Step 4: Implement the remote interface method(s)

```
public double divide(double d1, double d2) throws RemoteException {
```

```
return d1/d2;
    }
```
The complete source code :
```
import java.rmi.*;
import java.rmi.activation.*;

public class ActivationServerImpl extends Activatable
implements ActivationServer {

public ActivationServerImpl(ActivationID id, MarshalledObject data) throws
RemoteException {
super(id, 0);
}

public double divide(double d1, double d2) throws RemoteException {
return d1/d2;
    }
}
```

21.4 The policy file

Write the following code and save the file as policy (without any extension), in the directory in which the other files of this example are saved.

```
grant {
      // Allow everything for now
      permission java.security.AllPermission;
};
```

21.5 Creating the "setup" class

The job of the "setup" class is to create all the information necessary for the activatable class, without necessarily creating an instance of the remote object.

The setup class passes the information about the activatable class to rmid, registers a remote reference (an instance of the activatable class's stub class) and an identifier (name) with the rmiregistry, and then the setup class may exit. There are seven steps to create a setup class:

Step 1: Make the appropriate imports in the setup class

```
import java.rmi.*;
import java.rmi.activation.*
import java.util.Properties;
```

Step 2: Install a SecurityManager

System.setSecurityManager(new RMISecurityManager());

Step 3: Create an ActivationGroup instance

In the setup application, the job of the activation group descriptor is to provide all the information that rmid will require to contact the appropriate existing JVM or spawn a new JVM for the activatable object.

```
Properties props = new Properties();
props.put("java.security.policy",
   "policy");

ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc exampleGroup = new ActivationGroupDesc(props, ace);

The following line registers the ActivationGroupDesc with the activation
system to obtain its ID:

ActivationGroupID agi =
   ActivationGroup.getSystem().registerGroup(exampleGroup);
```

The following line creates the group:

ActivationGroup.createGroup(agi, exampleGroup, 0);

Step 4: Create an ActivationDesc **instance**

The job of the activation descriptor is to provide all the information that rmid will require to create a new instance of the implementation class.

The following line specifies the location where the class definition is available when this object is activated. The trailing slash "/" at the end of the URL is very important to locate the

classes. In this example we assume that the class files are located in the "activation" directory in the C drive.

String location = "file:/c/activation/";

The following line defines the other parameter required for the ActivationDesc Constructor:

MarshalledObject data = null;

The ActivationDesc constructor is called as shown in the following line:

ActivationDesc desc = new ActivationDesc ("ActivationServerImpl",
                        location, data);

Step 5: Declare an instance of your remote interface and register the activation descriptor with rmid

ActivationServer mri = (ActivationServer)Activatable.register(desc);
System.out.println("Got the stub for the Activation Server");

Step 6: Bind the stub, that was returned by the Activatable.register method, to a name in the rmiregistry

Naming.rebind("ActivationServerImpl", mri);
System.out.println("Exported Activation Server");

Step 7: Quit the setup application

System.exit(0);

The Complete source code :

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class Setup {
```

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


System.setSecurityManager(new RMISecurityManager());
Properties props = new Properties();
props.put("java.security.policy","policy");
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc exampleGroup = new
ActivationGroupDesc(props, ace);
ActivationGroupID agi =
ActivationGroup.getSystem().registerGroup(exampleGroup);
ActivationGroup.createGroup(agi, exampleGroup, 0);
String location = "file:/c/activation/";
MarshalledObject data = null;
ActivationDesc desc = new ActivationDesc
("ActivationServerImpl", location, data);
ActivationServer mri =
 (ActivationServer)Activatable.register(desc);
System.out.println("Got the stub for the Activation Server");
Naming.rebind("ActivationServerImpl", mri);
System.out.println("Exported Activation Server");
System.exit(0);
    }
}
```

21.6 Compile and run the code

The complete source code for the client is given below.


```
import java.rmi.*;


public class DivideClient {
public static void main(String args[]) {


  try {
System.setSecurityManager(new RMISecurityManager());
String divideServerURL;
      divideServerURL="rmi://" + args[0] + "/ActivationServerImpl";
ActivationServer server;
server=(ActivationServer )Naming.lookup(divideServerURL);


System.out.println("The first number is: " + args[1]);
      double d1=Double.valueOf(args[1]).doubleValue();
```

```
        System.out.println("The second number is: " + args[2]);
        double d2=Double.valueOf(args[2]).doubleValue();


double result = server.divide(d1,d2);
        System.out.println("The result is: "+ result);
   } catch(Exception ex)  {
     ex.printStackTrace();
   }
 }
 }
```

There are six steps to compile and run the code:

Step 1: Compile the remote interface, implementation, client and setup classes

% javac -d . ActivationServer.java

% javac -d . ActivationServerImpl.java

% javac -d . DivideClient.java

% javac -d . Setup.java

Step 2: Run rmic **on the implementation class**

% rmic -d . ActivationServerImpl

Step 3: Start the rmiregistry

D:\ rmiregistry &

> **Note:** Before the rmiregistry is started, the window in which it is run, should either have no CLASSPATH set or should have a CLASSPATH that does not include the path to any classes that should be downloaded to the client, including the stubs for the remote object implementation classes.
>
> If the rmiregistry finds the stub classes in its CLASSPATH, it will ignore the server's java.rmi.server.codebase property, and as a result, the client(s) will not be able to download the stub code for the remote object.

Step 4: Start the activation daemon, rmid

D:\rmid &

Step 5: Run the setup program

Run the setup, setting the codebase property to be the location of the implementation stubs. There are four things that need to go on the same command line:

1. The "java" command
2. A property name=value pair that specifies the location of the security policy file
3. A property to specify where the stub code lives (no spaces from the "-D" all the way though the last "/")
4. The fully-qualified package name of the setup program.

There should be one space just after the word "java", one between the two properties, and a third one just before the word "examples" (which is very hard to see when you view this as text, in a browser, or on paper).

%   java   -Djava.security.policy=/home/rmi_tutorial/activation/policy   -Djava.rmi.server.codebase=file:/home/rmi_tutorial/activation/ examples.activation.Setup

The codebase property will be resolved to a URL, so it must have the form "http://aHost/somesource/" or "file:/myDirectory/location/" or, due to the requirements of some operating systems, "file:///myDirectory/location/" (three slashes after the "file:").

While a file: URL is sometimes easier to use for running example code, using the file: URL will mean that the only clients that will be able to access the server are those that can access the same files system as the server (either by virtue of running on the same machine as the server or by using a shared filesystem, such as NFS).

Please note that each of these sample URL strings has a trailing "/". The trailing slash is a requirement for the URL set by the java.rmi.server.codebase property, so the implementation can resolve (find) your class definition(s) properly.

If you forget the trailing slash on the property, or if the class files can't be located at the source (they aren't really being made available for download) or if you misspell the property name, you'll get thrown a **java.lang.ClassNotFoundException**. This exception will be thrown when you try to bind your remote object to the rmiregistry, or when the first client attempts to

access that object's stub. If the latter case occurs, you have another problem as well because the rmi registry was finding its files in the classpath.

The server output should look like this:

Got the stub for the ActivatableImplementation

Exported ActivatableImplementation

Step 6:   Run the client

The argument to the DivideClient program is the hostname of the implementation server.

% java -Djava.security.policy=/home/policy DivideClient

21.7 Brain Storm

1.   Compare RMI with other Middleware Specifications?
2.   What makes Java-RMI tick?
3.   I have local objects that are synchronized. When I make them remote, my application hangs. What's the problem?
4.   Why do I get an exception for an unexpected hostname and/or port number when I call Naming.lookup?
5.   Why do Naming.bind and Naming.lookup take an extraordinarily long time on Windows?

જીભ્ય

`

Lecture 22

Discussion

Lecture 23

Servlets Introduction

Objectives

In this lecture you will learn the following

- Common Gateway Interface

- Java Servlet API

- Servlets and Advantages

- Servlets Over CGI

23.1 Snap Shot - Introduction

The Internet, with particular respect to the World Wide Web, is growing at a tremendous rate. With over one million new pages going live every day, each one vying for our attention, greater emphasis is being placed on the servers delivering this information. The Internet is no longer a colorful brochure, where the user merely flicks from page to page. It has become a fully interactive experience, complete with inline video and stereo sound.

The raise of server-side applications is one of the most exciting trends in Java programming. Java servlets are a key component of server-side Java development.

A servlet is a small, pluggable extension to a server that enhances the server's functionality. These servlets are commonly used with web servers, where they can take the place of CGI scripts.

Servlets are to the server what applets are to the client. They extend the functionality of the web server in much the same way an applet extends the browser. To demonstrate the power of servlets, Sun developed the Java Web Server using the Java Server API. This is a complete web server, supporting both servlets and CGI, written entirely in Java. Implementing the web server in Java allows the web server to run on any platform with a Java Virtual Machine (JVM).

23.1.1 Common Gateway Interface (CGI)

The Common Gateway Interface popularly known as CGI, was one of the first techniques used to create dynamic content. A web server passes certain request to the external program with the help of CGI. The advent of CGI made it possible to create all sorts of new functionality in web pages. CGI defines a standard for communication between the web server and a separate program running on the same machine. CGI doesn't define anything else, and as a consequence, CGI scripts may be implemented in any language the platform will run.

This lack of standardization at the server has meant CGI scripts implemented for one server platform have had to be redeveloped when required to run on another web server, on another platform.

Another problem area within CGI-based solutions is their overall efficiency. Every client request that requires processing by a CGI script spawns a separate program instance. This takes time. The operating system has to load the program, allocate memory for the program, and then deallocate and unload the program from memory. While the operating system is performing the housekeeping, nothing else can run. This is known as a heavyweight context switch. This is the reason CGI scripts are not suitable for applications that receive many client requests.

23.1.2 Java Server API

The foundation of the Server Toolkit is the Java Server API. This API allows the building of complete server-side applications. A servlet merely extends the functionality of a server, and using this API, the server may be built. The best example of a server built using this API is the Java Web Server from Sun. Although it is a very powerful and feature-rich API, it is not expected to be useful for everyone; instead, many developers will be more interested in the servlet API.

23.1.3 Java Servlet API

The servlet API allows the development of servlets. A servlet is designed to extend the functionality of the server it is running under. The servlet API comes as a set of classes that are used to form the base class for any user servlets. Any server supporting the servlet API will run any servlets developed using the servlet toolkit.

23.2 Servlet Overview

23.2.1 Java Servlets

Servlets are Java technology's answer to CGI programming. They are programs that run on a Web server and build Web pages. Building Web pages on the fly is useful (and commonly done) for a number of reasons:

- The Web page is based on data submitted by the user. For example the results pages from search engines are generated this way, and programs that process orders for e-commerce sites do this as well.

- The data changes frequently. For example, a weather-report or news headlines page might build the page dynamically, perhaps returning a previously built page if it is still up to date.

- The Web page uses information from corporate databases or other such sources. For example, you would use this for making a Web page at an on-line store that lists current prices and number of items in stock.

23.2.2 What is the Advantage of Servlets Over "Traditional" CGI?

Java Servlets are more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI and than many alternative CGI-like technologies. (More importantly, Servlet developers get paid more than Perl programmer's get-).

- **Efficient.** With traditional CGI, a new process is started for each HTTP request. If the CGI program does a relatively fast operation, the overhead of starting the process can dominate the execution time. With Servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are N simultaneous request to the same CGI program, then the code for the CGI program is loaded into memory N times. With Servlets, however, there are N threads but only a single copy of the Servlet class. Servlets also have more alternatives than do regular CGI programs for optimizations such as caching previous computations, keeping database connections open, and the like.

- **Convenient.** If you already know Java, why learn Perl too? Besides the convenience of being able to use a familiar language, Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.

- **Powerful.** Java Servlets let you easily do several things that are difficult or impossible with regular CGI. For one thing, Servlets can talk directly to the Web server (regular CGI

programs can't). This simplifies operations that need to look up images and other data stored in standard places. Servlets can also share data among each other, making useful things like database connection pools easy to implement. They can also maintain information from request to request, simplifying things like session tracking and caching of previous computations.

- **Portable.** Servlets are written in Java and follow a well-standardized API. Consequently, Servlets written for, say I-Planet Enterprise Server can run virtually unchanged on Apache, Microsoft IIS, or Webster. Servlets are supported directly or via a plugging on almost every major Web server.

- **Inexpensive.** There are a number of free or very inexpensive Web servers available that are good for "personal" use or low-volume Web sites. However, with the major exception of Apache, which is free, most of the commercial-quality Web servers are relatively expensive. Nevertheless, once you have a Web server, no matter the cost of that server, adding Servlet support to it (if it doesn't come reconfigured to support Servlets) is generally free or cheap.

23.3 Starting with Servlets

23.3.1 Basic Servlet Structure

Here's the outline of a basic servlet that handles GET requests. GET requests, for those unfamiliar with HTTP, are requests made by browsers when the user types in a URL on the address line, follows a link from a Web page, or makes an HTML form that does not specify a METHOD. Servlets can also very easily handle POST requests, which are generated when someone creates an HTML form that specifies METHOD="POST". We'll discuss that in later sections.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SomeServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
```

```
// Use "request" to read incoming HTTP headers (e.g. cookies)
// and HTML form data (e.g. data the user entered and        submitted)

// Use "response" to specify the HTTP response line and      headers
// (e.g. specifying the content type, setting cookies).

   PrintWriter out = response.getWriter();
// Use "out" to send content to browser
  }
}
```

To be a Servlet, a class should extend <u>HttpServlet</u> and override doGet or doPost (or both), depending on whether the data is being sent by GET or by POST. These methods take two arguments: an <u>HttpServletRequest</u> and an <u>HttpServletResponse</u>. The HttpServletRequest has methods that let you find out about incoming information such as FORM data, HTTP request headers, and the like. The HttpServletResponse has methods that lets you specify the HTTP response line (200, 404, etc.), response headers (Content-Type, Set-Cookie, etc.), and, most importantly, lets you obtain a PrintWriter used to send output back to the client. For simple Servlets, most of the effort is spent in println statements that generate the desired page. Note that doGet and doPost throw two exceptions, so you are required to include them in the declaration. Also note that you have to import classes in java.io (for PrintWriter, etc.), javax.servlet (for HttpServlet, etc.), and javax.servlet.http (for HttpServletRequest and HttpServletResponse). Finally, note that doGet and doPost are called by the service method, and sometimes you may want to override service directly, e.g. for a Servlet that handles both GET and POST request.

23.3.2 The Life Cycle of a Servlet

Since servlets come in the form of Java objects, there are many different variations on not only how they are loaded but also how they are unloaded again, if at all. When the server decides a particular servlet is to be loaded, it uses the standard Java class loading mechanisms to create the class instance. Using this technique, servlets can be loaded from anywhere on the network, and if the server is connected to the Internet, from anywhere in the world.

Class c = Class.forName("http://<server>/testclass");

Once this method has returned, the class can be accessed as normal. Asuming the URL doesn't upset the security of the system, classes can be located anywhere on the network.

> **Note:** Loading classes from within a Java servlet, applet, or any Java program is achieved through the java.lang.Class class. This class method returns the class object associated with the full URL that was passed in.

Forgetting about the potential security implications this may have (these are addressed in the next section on servlet security) this provides a great advantage to distribution and resource sharing. Unlike the applet-loading model, where every client running the applet loads the applet from the same origin, the servlet has the ability to load from multiple hosts. For example, as shown in Figure 7.1, one server can run the servlet code that originated from another.
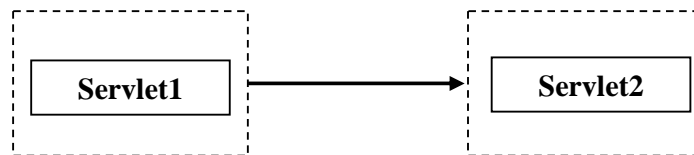
Figure 23.1 Locating servlets

The actual loading and execution of the servlet is straightforward enough--when a client connection is accepted and the servlet is not in memory, it is loaded. However, a servlet can be loaded if it has not yet been addressed. A servlet can be loaded either at server startup or dynamically when it is accessed.

The Java Web Server has an administration section that allows the administrator to specify which servlets are loaded at startup. This allows the servlet to be ready in memory forthe first client request to come in. Generally, only servlets that are expected to be heavily usedare loaded at server startup. Loading the servlet at startup ensures the response time for all requests is kept to a minimum, as opposed to waiting for the servlet to be loaded.

Alternatively, the servlet can be loaded when it is first accessed. No matter how it is loaded, all servlets follow the same cycle as shown in Figure 7.2.
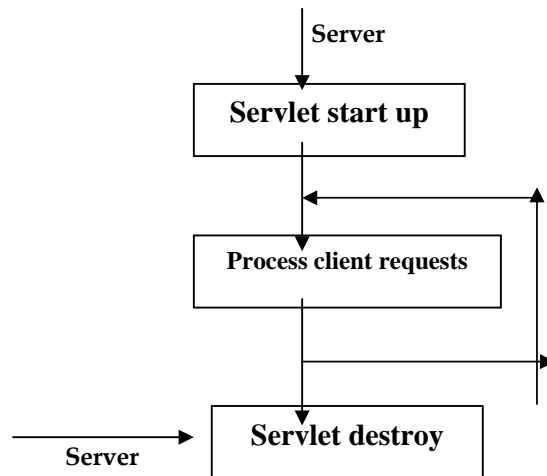
Figure 23.2 Servlet cycle

When the servlet is first loaded, a single method is called that can be used to initialize any startup data. For example, if the servlet is to be used for accessing a database, then the database connection could be opened from this method. Since the database connection need be opened only once, this is the perfect place for it.

Once the servlet class object has been initialized, it is ready to serve client requests. Servicing a client request is performed by the server (or another servlet, if they are in a chain) calling a predefined method for servicing requests. Since the servlet will be used in a multithreaded environment, the method servicing the request has to be made thread-safe, assuming shared variables are being accessed. For example, assume the servlet begins to service one client request, and halfway through it, another request comes in. If the first instance of the servlet had set any global variables, then it is possible for the second instance to change them. The first one will then read the wrong values.

Servlets are generally unloaded and removed from memory only when the server shuts down. However, there are situations when a servlet is unloaded immediately after completing a client request; some server-side includes servlets are good examples. This will be detailed in the server-side includes .

23.3.3 Servlet Security

Servlets are implemented in Java, so consequently, they benefit from all of the security features offered by Java. Java servlets come in two different modes: trusted and untrusted. A

trusted program, such as a Java application, has the same access to the system as any other program, including the following:

- Read/write files on the local machine
- Open connection to any host on the network
- Execution of other programs

Security features are controlled through the SecurityManager class of the run time environment. This determines whether a class has sufficient access rights to use a particular file, device, or program.

> **Note:** Java, unlike conventional programming languages, has a great deal of security built into the language itself, even before SecurityManager is called upon for its services. A Java program, whether it be an applet, servlet, or any other code variation, cannot access memory directly. Casting an integer to a memory pointer and accessing it simply cannot be done. By disallowing this action, the Java code cannot eavesdrop on other programs or corrupt areas of memory. The benefit of this is if for some reason a Java program crashes, no other program in the system is affected.

All of these security features are not unique to Java servlets only, but to all Java programs. As handy as these features are, they can severely hamper functionality in some instances. Imagine a servlet that was designed to process HTML forms by storing them in a file. If the SecurityManager did not permit writes to occur, then it would be impossible to create the file. But before you start reaching for the CGI book, know that help is at hand.

To understand why Java has such tight security when compared to other alternative technologies such as CGI and ActiveX, you have to look at the context in which the applications will be running. One of the goals of Java is to build a completely open, distributed-computing model which will allow users to download and run code on demand, as opposed to installing the code beforehand. On the face of it, this is not a bad idea, but do you really want to download and execute a program that has the potential of reading your personal files, or, even worse, reformatting your hard disk?

In order to gain the trust of users, the makers of Java had to build in very tight security features that could not be compromised. The reason for this need-to-know type of protection is that the user cannot be sure of a developer's intentions. Maybe they are a trustworthy organization and won't mess up a user's hard disk, but on the other hand, they may be a consortium of hackers masquerading behind a legitimate company, out to steal information. The user has no way of knowing.

For this reason, all Java programs coming from an outside resource, such as the Internet, are considered untrusted. Can Java programs become trusted? The short answer is yes.

A trusted program is one the user feels won't do anything it shouldn't do. For example, servlets definitely need to have the ability to read and write files if they are to serve any useful function in the areas of HTML processing. A servlet can be trusted by the server if it is or has one of the following:

- Built-in servlets
- Digital signatures

Servlets that are built-in are those which have been verified by the administrator and allow the server to load them with full permission. Generally, this is achieved by giving the server a list of servlets that are considered built-in or trusted; this list is checked every time a servlet is requested. If a request is made for a servlet that doesn't appear as part of this list, the servlets access rights are severely restricted to the status of an applet running in the client browser.

The other method for creating a trusted servlet is to digitally sign each servlet. Before running the code, the server can check to see if the signature is from a known, trusted source before granting full access to it.

Note: Simply adding a digital signature doesn't prevent attacks. For example, a servlet developer with a known, trusted signature may wake up one morning, think, "What the heck?" and then develop a rogue piece of software that completely erases the system directory.

Some technologies, ActiveX for example, completely rely on this type of security. One of the most famous examples of bad security is the Internet Exploder, which was an ActiveX plug in. Once it was loaded, it shut down the client's machine without warning. The signature, or certificate as it is known in the ActiveX world, was attained legitimately. When the developer was challenged, the reply was that a new Internet-based utility had been developed to allow users to turn off their machines.

By implementing the previously discussed security systems, administrators can sleep a little easier at night, safe in the knowledge that their servers aren't being taken over by rogue servlets.

23.3.4 A Simple Servlet Generating Plain Text

Here is a simple Servlet that just generates plain text. The following section will show the more usual case where HTML is generated.

HelloWorld.java

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("Hello World");
  }
}
```

Compiling and installing the Servlet

Note that the specific details for installing Servlets vary from Web server to Web server. Please refer to your Web server documentation for definitive directions. The on-line examples running on Java Web Server (JWS) 2.0, where Servlets are expected to be in a directory called Servlets in the JWS installation hierarchy. However, Servlet is placed in a separate package (hall) to avoid conflicts with other Servlets on this server; you'll want to do the same if you are using a Web server that is used by other people and doesn't have a good infrastructure for "virtual servers" to prevent these conflicts automatically. Thus, HelloWorld.java actually goes in a subdirectory called hall in the Servlets directory. Note that setup on most other servers is similar, material.

Setting Class.Java

One way is to set your CLASSPATH to point to the directory above the one actually containing your Servlets. You can them compile normally from within the directory. For example, if your base directory is C:\JavaWebServer\servlets and your package name (and thus subdirectory name) is hall, and you were on Windows, you'd do:

DOS> set CLASSPATH=C:\JavaWebServer\servlets;%CLASSPATH%

DOS> cd C:\JavaWebServer\servlets\hall DOS> javac YourServlet.java

The first part, setting the CLASSPATH, you probably wants to do permanently, rather than each time you start a new DOS window. On Windows 95/98 you'd typically put the "set CLASSPATH=..." statement in your autoexec.bat file somewhere after the line that set the CLASSPATH to point to servlet.jar and jsp.jar. On Windows NT, you'd go to the Start menu, select Settings, select Control Panel, select System, select Environment, then enter the variable and value. Note also that if your package were of the form name1.name2.name3 rather than simply name1 as here, you'd still have the CLASSPATH point to the top-level directory of your package hierarchy (the one containing name1).

A second way to compile classes that are in packages is to go to the directory above the one containing your Servlets, and then do "javac directory\YourServlet.java" (Windows; note the backslash) or "javac directory/YourServlet.java" (Unix; note the forward slash). For example, suppose again that your base directory is C:\JavaWebServer\servlets and your package name (and thus subdirectory name) is hall, and you were on Windows. In that case, you'd do the following:

DOS> cd C:\JavaWebServer\servlets

DOS> javacprogs\YourServlet.java

Note that, on Windows, most JDK 1.1 versions of javac require a backslash, not a forward slash, after the directory name. This is fixed in JDK 1.2, but since many Web servers are configured to use JDK 1.1, many Servlet authors stick with JDK 1.1 for portability. Finally, another advanced option is to keep the source code in a location distinct from the .class files, and use javac's "-d" option to install them in the location the Web server expects.

Running the Servlet

With the Java Web Server, Servlets are placed in the Servlets directory within the main JWS installation directory, and are invoked via http://host/servlet/ServletName. Note that the directory is Servlets, plural, while the URL refers to Servlet, singular. Since this example was placed in the progs package, it would be invoked via http://host/servlet/progs.HelloWorld. Other Web servers may have slightly different

conventions on where to install Servlets and how to invoke them. Most servers also let you define aliases for Servlets, so that a Servlet can be invoked via http://host/any-path/any-file.html. The process for doing this is Completely server-specific.

Figure 23.**3** A Simple Servlet

23.3.5 A Servlet that Generates HTML

Most Servlets generate HTML, not plain text as in the previous example. To do that, you need two additional steps: tell the browser that you're sending back HTML, and modify the println statements to build a legal Web page. The first step is done by setting the Content-Type response header. In general, headers can be set via the setHeader method of HttpServletResponse, but setting the content type is such a common task that there is also a special setContentType method just for this purpose. Note that you need to set response headers before actually returning any of the content via the PrintWriter. Here's an example:

Hello WWW. java

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0"+
```

Centre for Information Technology and Engineering, Manonaniam Sundaranar University

```
                                        "Transitional//EN\">\n" +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Hello WWW</H1>\n" +
                "</BODY></HTML>");
    }
}
```
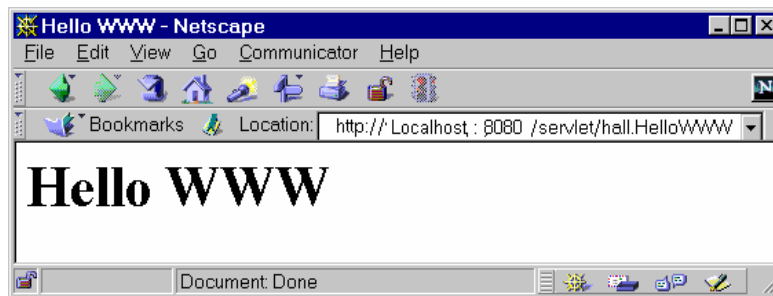
Hello WWW Result

Figure 23.4 – Hello www result

Simple HTML-Building Utilities

It is a bit cumbersome to generate HTML with println statements. The real solution is to use Java Server Pages (JSP), which is discussed in later Chapters. However, for standard Servlets, there are two parts of the Web page (DOCTYPE and HEAD) that are unlikely to change and thus could benefit from being incorporated into a simple utility file.

The DOCTYPE line is technically required by the HTML spec, and although most major browsers ignore it, it is very useful when sending pages to formal HTML validators. These validators compare the HTML syntax of pages against the formal HTML specification, and use the DOCTYPE line to determine which version of HTML to check against. Their use is very highly recommended both for static HTML pages and for pages generated via Servlets, so the use of DOCTYPE is well worth the effort, especially if it can be easily incorporated into a Servlet utilities class.

In many Web pages, the HEADline contains nothing but the TITLE, although advanced developers may want to include META tags and style sheets. But for the simple case.

The example below

create a method that takes a title as input and returns the DOCTYPE, HEAD, and TITLE entries as output. Here's the code:

Servlet Utilities. java

```
package hall;

public class ServletUtilities {
  public static final String DOCTYPE =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">";

  public static String headWithTitle(String title) {
    return(DOCTYPE + "\n" +
           "<HTML>\n" +
           "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
  }

  // Other utilities will be shown later...
}
```

HelloWWW2.java

```
Here's a rewrite of the HelloWWW class that uses this.
import hall.ServletUtilities;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW2 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
out.println(ServletUtilities.headWithTitle("Hello WWW") +
               "<BODY>\n" +
               "<H1>Hello WWW</H1>\n" +
               "</BODY></HTML>");
  }
}
```

23.4 Short Summary

- A servlet is a small, pluggable extension to a server that enhances the server's functionality.

- Java Servlets are more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI…

- CGI was one of the first techniques used to create dynamic content.

- To be a Servlet, a class should extend <u>HttpServlet</u> and override doGet or doPost (or both), depending on whether the data is being sent by GET or by POST.

- The specific details for installing Servlets vary from Web server to Web server.

23.5 Brain Storm

1. Write short on the following
      a. Common Gateway Interface (CGI)
      b. Java Server API
      c. Java Servlet API
2. Write short note on servlet.
3. Explain servlet structure.

ℰᏜ

Lecture 24

Request Headers

Objectives

In this lecture you will learn the following...

- ✎ Handling form data

- ✎ About Request Headers

- ✎ Objectives

<div style="border:2px solid black; padding:10px;">

Coverage Plan

</div>

Lecture 24

24.1 Snap Shot - Handling form Data

If you've ever used a Web search engine, visited an on-line bookstore, tracked stocks on-line, or asked a Web-based site for quotes on plane tickets, you've probably seen funny looking URLs like http://host/path?user=Marty+Hall&origin=bwi&dest=lax. The part after the question mark (i.e. user=Marty+Hall&origin=bwi&dest=lax) is known as form data, and is the most common way to get data from a Web page to a server-side program. It can be attached to the end of the URL after a question mark (as above), for GET requests, or sent to the server on a separate line, for POST requests. Extracting the needed information from this form data is traditionally one of the most tedious parts of CGI programming. First of all, you have to read the data one way for GET requests (in traditional CGI, this is usually via the QUERY_STRING environment variable), and another way for POST requests (usually by reading the standard input).

Second, you have to chop the pairs at the ampersands, then separate the parameter names (left of the equal sign) from the parameter values (right of the equal sign).

 Third, you have to URL-decode the values. Alphanumeric characters get sent unchanged, but spaces get converted to plus signs and other characters get converted to %XX where XX is the ASCII (or ISO Latin-1) value of the character, in hex. For example, if someone entered a value of "~hall, ~gates, and ~mcnealy" into a textfield with the name "users" in an HTML form, the data would get sent as "users=%7Ehall%2C+%7Egates%2C+and+%7Emcnealy".

Finally, the fourth reason that parsing form data is tedious is that values can be omitted (e.g. param1=val1&param2=&param3=val3) and a parameter can have more than one value in that the same parameter can appear more than once (e.g. param1=val1&param2=val2&param1=val3).

One of the nice features of Java Servlets is that all of this form parsing is handled automatically. You simply call the getParameter method of the HttpServletRequest, supplying the parameter name as an argument. Note that parameter names are case sensitive. You do this exactly the same way when the data is sent via GET as you do when it is sent via POST. The return value is a String corresponding to the undecoded value of the first occurrence of that parameter name. An empty String is returned if the parameter exists but has no value, and null is returned if there was no such parameter. If the parameter could

potentially have more than one value, as in the example above, you should call getParameterValues instead of getParameter. This returns an array of strings. Finally, although in real applications your Servlets probably have a specific set of parameter names they are looking for, for debugging purposes it is sometimes useful to get a full list. Use getParameterNames for this, which returns an Enumeration, each entry of which can be cast to a String and used in a getParameter call.

Example: Reading Three Parameters

Three Params. Java

Here's a simple example that reads parameters named param1, param2, and param3, listing their values in a bulleted list.

Note: although you are required to specify response settings (content type, status line, and other HTTP headings) before beginning to generate the content, there is no requirement that you read the request parameters at any particular time. Also you can easily make Servlets that can handle both GET and POST data, simply by having its doPost method call doGet or by overriding service (which calls doGet, doPost, doHead, etc.).

This is good standard practice, since it requires very little extra work and permits flexibility on the part of the client. If you're used to the traditional CGI approach where you read POST data via the standard input, you should note that there is a similar way with Servlets by first calling getReader or getInputStream on the HttpServletRequest. This is a bad idea for regular parameters, but might be of use for uploaded files or POST data being sent by custom clients rather than via HTML forms.

Note: however, that if you read the POST data in that manner, it might no longer be found by getParameter.

```
import hall.ServletUtilities;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Reading Three Request Parameters";
```

```
        out.println(ServletUtilities.headWithTitle(title) +
                "<BODY>\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI>param1: "
                + request.getParameter("param1") + "\n" +
                "  <LI>param2: "
                + request.getParameter("param2") + "\n" +
                "  <LI>param3: "
                + request.getParameter("param3") + "\n" +
                "</UL>\n" +
                "</BODY></HTML>");
    }

    public void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
      doGet(request, response);
    }
}
```

Program to create HTML page to accept user input.

```
        <html>
        <body>
<form method="post" action="http://localhost:8080/servlet/ThreeParams">
        firstName:<input name="param1"><br>
        lastName:<input name="param2"><br>
        password:<input name="param3"><br>
        <input type=submit>
        </form>
        </body>
        </html>
```
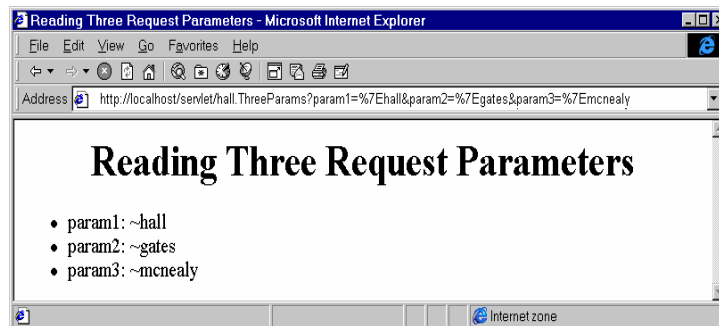
Three Params Output

Figure 24.1 Three Params Output

Example: Listing All Form Data

Here's an example that looks up all the parameter names that were sent and puts them in a table. It highlights parameters that have zero values as well as ones that have multiple values. First, it looks up all the parameter names via the getParameterNames method of HttpServletRequest. This returns an Enumeration. Next, it loops down the Enumeration in the standard manner, using hasMoreElements to determine when to stop and using nextElement to get each entry. Since nextElement returns an Object, it casts the result to a String and passes that to getParameterValues, yielding an array of Strings. If that array is one entry long and contains only an empty string, then the parameter had no values, and the Servlet generates an italicized "No Value" entry. If the array is more than one entry long, then the parameter had multiple values, and they are displayed in a bulleted list. Otherwise the one main value is just placed into the table.

Show Parameters. java

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import hall.ServletUtilities;


/** Shows all the parameters sent to the Servlet via either
 *  GET or POST. Specially marks parameters that have no values or
 *  multiple values.
 */

public class ShowParameters extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Reading All Request Parameters";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
"<TH>Parameter Name<TH>Parameter Value(s)");
```

```
      Enumeration paramNames = request.getParameterNames();
      while(paramNames.hasMoreElements()) {
        String paramName = (String)paramNames.nextElement();
        out.println("<TR><TD>" + paramName + "\n<TD>");
        String[] paramValues = request.getParameterValues(paramName);
        if (paramValues.length == 1) {
          String paramValue = paramValues[0];
          if (paramValue.length() == 0)
            out.print("<I>No Value</I>");
          else
            out.print(paramValue);
        } else {
          out.println("<UL>");
          for(int i=0; i<paramValues.length; i++) {
            out.println("<LI>" + paramValues[i]);
          }
          out.println("</UL>");
        }
      }
    out.println("</TABLE>\n</BODY></HTML>");
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Front End to Show Parameters

Here's an HTML form that sends a number of parameters to this Servlet. It uses POST to send the data (as should all forms that have PASSWORD entries), demonstrating the value of having Servlets include both a doGet and a doPost.

PostForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>A Sample FORM using POST</TITLE>
</HEAD>
```

```
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">A Sample FORM using POST</H1>

<FORM ACTION="/servlet/ ShowParameters"
      METHOD="POST">
  Item Number:
  <INPUT TYPE="TEXT" NAME="itemNum"><BR>
  Quantity:
  <INPUT TYPE="TEXT" NAME="quantity"><BR>
  Price Each:
  <INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>
  <HR>
  First Name:
  <INPUT TYPE="TEXT" NAME="firstName"><BR>
  Last Name:
  <INPUT TYPE="TEXT" NAME="lastName"><BR>
  Middle Initial:
  <INPUT TYPE="TEXT" NAME="initial"><BR>
  Shipping Address:
  <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
  Credit Card:<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                    VALUE="Visa">Visa<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                    VALUE="Master Card">Master Card<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                    VALUE="Amex">American Express<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                    VALUE="Discover">Discover<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                    VALUE="Java SmartCard">Java SmartCard<BR>
  Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
  Repeat Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
  <CENTER>
    <INPUT TYPE="SUBMIT" VALUE="Submit Order">
  </CENTER>
</FORM>
</BODY>
</HTML>
```
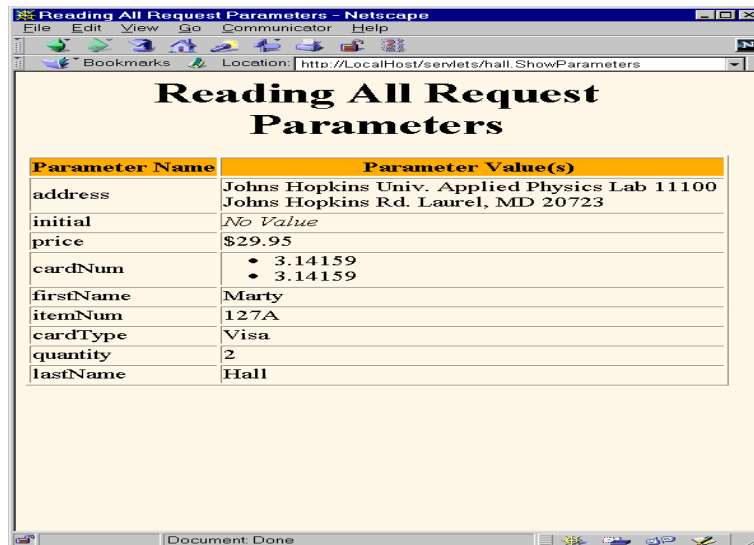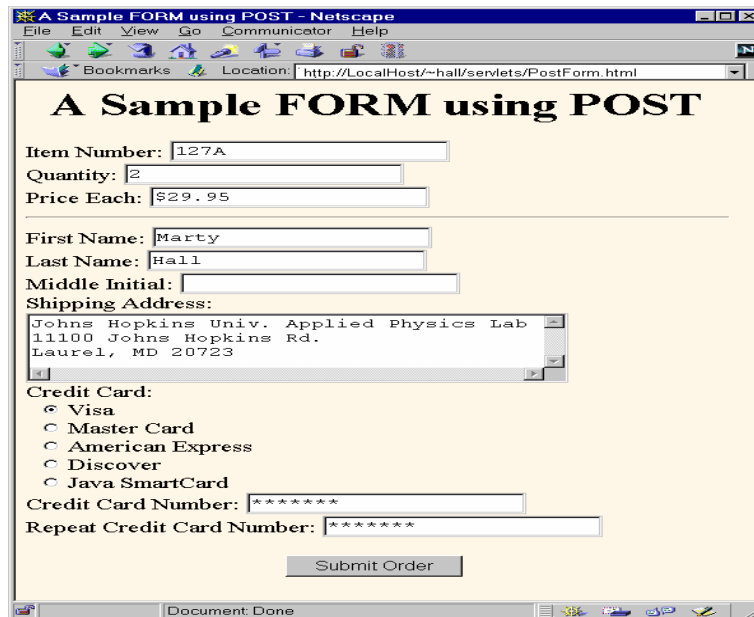
Figure 24.2 A sample form using posr

Figure 24.3 Submission Result

24.2 Request Headers

24.2.1 An Overview of Request Headers

When an HTTP client (e.g. a browser) sends a request, it is required to supply a request line (usually GET or POST). If it wants to, it can also send a number of headers, all of, which are

optional except for Content-Length, which is required only for POST requests. Here are the most common headers:

- Accept the MIME types the browser prefers.

- Accept-Chariest the character set the browser expects.

- Accept-Encoding the types of data encoding (such as gzip) the browser knows how to decode. Servlets can explicitly check for gzip support and return gripped HTML pages to browsers that support them, setting the Content-Encoding response header to indicate that they are gripped. In many cases, this can reduce page download times by a factor of five or ten.

- Accept-Language-The language the browser is expecting, in case the server has versions in more than one language.

- Authorization- Authorization info, usually in response to a WWW-Authenticate header from the server.

- Connection Use persistent connection? If a Servlet gets a Keep-Alive value here, or gets a request line indicating HTTP 1.1 (where persistent connections are the default), it may be able to take advantage of persistent connections, saving significant time for Web pages that include several small pieces (images or applet classes). To do this, it needs to send a Content-Length header in the response, which is most easily accomplished by writing into a Byte Array Output Stream, then looking up the size just before writing it out.

- Content-Length (for POST messages, how much data is attached)

- Cookie (one of the most important headers;)

- From (email address of requester; only used by Web spiders and other custom clients, not by browsers)

- Host (host and port as listed in the original URL)

- If-Modified-Since (only return documents newer than this, otherwise send a 304 "Not Modified" response)

- Parma (the no-cache value indicates that the server should return a fresh document, even if it is a proxy with a local copy)

- Referrer (the URL of the page containing the link the user followed to get to current page)

- User-Agent (type of browser, useful if Servlet is returning browser-specific content)

- UA-Pixels, UA-Color, UA-OS, UA-CPU (nonstandard headers sent by some Internet Explorer versions, indicating screen size, color depth, operating system, and CPU type used by the browser's system)

24.2.2 Reading Request Headers from Servlets

Reading headers is very straightforward; just call the getHeader method of the HttpServletRequest, which returns a String if the header was supplied on this request, null otherwise. However, there are a couple of headers that are so commonly used that they have special access methods. The getCookies method returns the contents of the Cookie header, parsed and stored in an array of Cookie objects. The getAuthType and getRemoteUser methods break the Authorization header into its component pieces. The getDateHeader and getIntHeader methods read the specified header and then convert them to Date and int values, respectively.

Rather than looking up one particular header, you can use the getHeaderNames to get an Enumeration of all header names received on this particular request. Finally, in addition to looking up the request headers, you can get information on the main request line itself. The getMethod method returns the main request method (normally GET or POST, but things like HEAD, PUT, and DELETE are possible). The getRequestURI method returns the URI (the part of the URL that came after the host and port, but before the form data). The getRequestProtocol returns the third part of the request line, which is generally "HTTP/1.0" or "HTTP/1.1".

24.2.3 Example: Printing all Headers

Here's a Servlet that simply creates a table of all the headers it receives, along with their associated values. It also prints out the three components of the main request line (method, URI, and protocol).

Show Request Headers. java

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import hall.ServletUtilities;

public class ShowRequestHeaders extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Servlet Example: Showing Request Headers";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<B>Request Method: </B>" +
                request.getMethod() + "<BR>\n" +
                "<B>Request URI: </B>" +
                request.getRequestURI() + "<BR>\n" +
                "<B>Request Protocol: </B>" +
                request.getProtocol() + "<BR><BR>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Header Name<TH>Header Value");
    Enumeration headerNames = request.getHeaderNames();
    while(headerNames.hasMoreElements()) {
      String headerName = (String)headerNames.nextElement();
      out.println("<TR><TD>" + headerName);
      out.println("    <TD>" + request.getHeader(headerName));
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }
```

```
    public void doPost(HttpServletRequest request,
                    HttpServletResponse response)
        throws ServletException, IOException {
      doGet(request, response);
    }
}
```

Show Request Headers Output

Here are the results of two typical requests, one from Netscape and one from Internet Explorer.
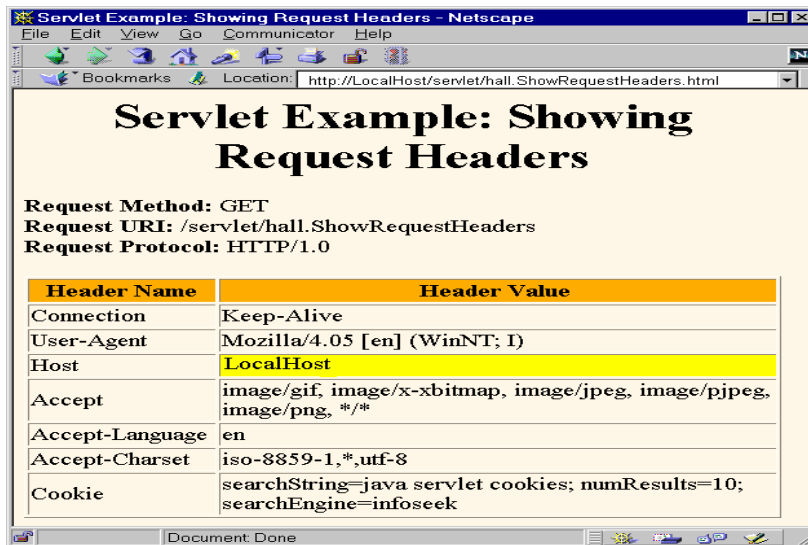
Figure 24.4 – Request header output
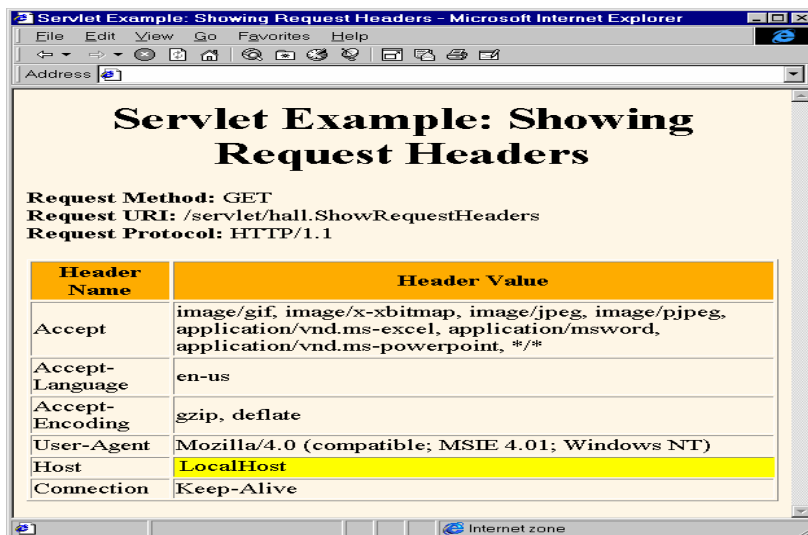
Figure 24.5 – Servlets example

24.3 Short summary

- Reading headers just call the getHeader method of the HttpServletRequest, to returns a String if the header was supplied on this request, null otherwise.

- The getCookies method returns the contents of the Cookie header, parsed and stored in an array of Cookie objects.

- The getAuthType and getRemoteUser methods break the Authorization header into its component pieces.

- The getDateHeader and getIntHeader methods read the specified header and then convert them to Date and int values, respectively.

- The getRequestURI method returns the URI

- The getRequestProtocol returns the third part of the request line, which is generally "HTTP/1.0" or "HTTP/1.1".

24.4 Brain Storm

1. Write short note on Request Headers.

2. How to read Request headers from Servlets

ℬℭ

Lecture 25

Response Headers

Objectives

In this lecture you will learn the following

- ✎ Introduction about response headers
- ✎ About response headers
- ✎ Some examples

<div style="border:2px solid black; text-align:center">Coverage Plan</div>

Lecture 25

25.1 Snap Shot - Response Headers

A response from a Web server normally consists of a status line, one or more response headers, a blank line, and the document. Setting the HTTP response headers often goes hand in hand with setting the status codes in the status line. For example, several of the "document moved" status codes have an accompanying Location header, and a 401 (Unauthorized) code must include an accompanying WWW-Authenticate header. However, specifying headers can play a useful role even when no unusual status code is set. Response headers can be used to specify cookies, to supply the modification date (for caching), to instruct the browser to reload the page after a designated interval, to say how long the file is so that persistent HTTP connections can be used, and many other tasks. The most general way to specify headers is by the setHeader method of HttpServletResponse, which takes two strings: the header name and the header value. Like setting the status codes, this must be done before any document content is sent. There are also two specialized methods to set headers that contain dates (setDateHeader) and integers (setIntHeader).

The first saves you the trouble of translating a Java date in milliseconds since the epoch (as returned by system.currentTimeMillis or the getTime method applied to a Date object) into a GMT time string.

The second spares you the minor inconvenience of converting an int to a String. Rather than setting a header outright, you can add a new header, in case a header with that name already exists. Use addHeader, addDateHeader, and addIntHeader for this.

Finally, HttpServletResponse also supplies a number of convenience methods for specifying common headers.

- The setContentType method sets the Content-Type header, and is used by the majority of Servlets.

- The setContentLength method sets the Content-Length header, useful if the browser supports persistent (keep-alive) HTTP connections.

- The addCookie method sets a cookie (there is no corresponding setCookie, since it is normal to have multiple Set-Cookie lines).

- The sendRedirect method sets the Location header as well as setting the status code to 302.

25.1.1 Common Response Headers and Their Meaning

Header	Interpretation/Purpose
Allow	What request methods (GET, POST, etc.) does the server support?
Content-Encoding	What method was used to encode the document? You need to decode it to get the type specified by the Content-Type header. Using gzip to compress the document can dramatically reduce download times for HTML files, but it is only supported by Netscape on Unix and IE 4 and 5 on Windows. On the other hand, gzipping HTML files can dramatically reduce download times, and Java's GZIPOutputStream makes it easy. So you should explicitly check if the browser supports this by looking at the Accept-Encoding header (i.e. via request.getHeader("Accept-Encoding")). That way, you can return gzipped pages to browser that know how to unzip them, but still return regular pages to other browsers.
Content-Length	How many bytes are being sent? This information is only needed if the browser is using a persistent (keep-alive) HTTP connection. If you want your servlet to take advantage of this when the browser supports it, your servlet should write the document into a ByteArrayOutputStream, look up its size when done, put that into the Content-Length field, then send the content via byteArrayStream.writeTo(response.getOutputStream()).
Content-Type	What is the MIME type of the following document? Default for Servlets is text/plain, but they usually explicitly specify text/html. Setting this header is so common that there is a special method in HttpServletResponse for it: setContentType.
Date	What is current time (in GMT)? Use the setDateHeader method to specify this header. That saves you the trouble of formatting the date string properly.
Expires	At what time should content be considered out of date and thus no longer cached?
Last-Modified	When was document last changed? Client can supply a date via an If-Modified-Since request header. This is treated as a conditional GET, with document only being returned if the Last-Modified date is later than the

	specified date. Otherwise a 304 (Not Modified) status line is returned. Again, use the setDateHeader method to specify this header.
Location	Where should client go to get document? This is usually set indirectly, along with a 302 status code, via the sendRedirect method of HttpServletResponse.
Refresh	How soon should browser ask for an updated page (in seconds)? Instead of just reloading current page, you can specify a specific page to load via setHeader("Refresh", "5; URL=http://host/path"). Note that this is commonly set via <META HTTP-EQUIV="Refresh" CONTENT="5; URL = http://host/path"> in the HEAD section of the HTML page, rather than as an explicit header from the server. This is because automatic reloading or forwarding is something often desired by HTML authors who do not have CGI or servlet access. But for Servlets, setting the header directly is easier and clearer. Note that this header means "reload this page or go to the specified URL in *N* seconds." It does **not** mean "reload this page or go to the specified URL *every N* seconds." So you have to send a Refresh header each time, and sending a 204 (No Content) status code stops the browser from reloading further, regardless of whether you explicitly send the Refresh header or use <META HTTP-EQUIV="Refresh" ...>. Note that this header is not officially part of HTTP 1.1, but is an extension supported by both Netscape and Internet Explorer.
Server	What server am I? Servlets don't usually set this; the Web server itself does.
Set-Cookie	Specifies cookie associated with page. Servlets should not use response.setHeader("Set-Cookie", ...), but instead use the special-purpose addCookie method of HttpServletResponse. See separate section on handling cookies.
WWW-Authenticate	What authorization type and realm should client supply in their Authorization header? This header is required in responses that have a 401 (Unauthorized) status line. E.g. response.setHeader("WWW-Authenticate", "BASIC realm=\"executives\""). Note that servlets do not usually handle this themselves, but instead let password-protected Web pages be handled by the Web server's specialized mechanisms (e.g. .htaccess).

Table 25.1 Common Response Headers and Their Meaning

25.1.2 Example: Automatically Reloading Pages as Content Changes

Here is an example that lets you ask for a list of some large prime numbers. Since this may take some time for very large numbers (e.g. 150 digits), the Servlet immediately returns the results found so far, but then keeps calculating, using a low-priority thread so that it won't degrade Web server performance. If the calculations are not complete, it instructs the browser to ask for a new page in a few seconds by sending it a Refresh header. Note that, in addition to illustrating the value of HTTP response headers, this example shows two other valuable Servlet capabilities.

- First, it shows that Servlets can handle multiple simultaneous connections, each in their own thread. In this case it maintains a Vector of previous requests for prime calculations, matching the current request to previous ones by looking at the number of primes (length of list) and number of digits (length of each prime), and synchronizing all access to this list.

- Secondly, it shows how easy it is for Servlets to maintain state between requests something that is cumbersome to implement in traditional CGI and many CGI alternatives. This lets the browser access the ongoing calculations when reloading the page, plus permits the Servlet to keep a list of the *N* most recently requested results, returning them immediately if a new request specifies the same parameters as a recent request.

PrimeNumbers.java

> **Note :** also uses **ServletUtilities.java**, shown earlier, **PrimeList.java** for creating a Vector of prime numbers in a background thread, and **Primes.java** for generating large random numbers of type BigInteger and checking if they are prime. (Code Listing for PrimeList.java and Primes.java is given in Listing 1 and Listing 2 below).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import hall.*;
```

```
public class PrimeNumbers extends HttpServlet {
  private static Vector primeListVector = new Vector();
  private static int maxPrimeLists = 30;
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
      throws ServletException, IOException {
    int numPrimes =
      ServletUtilities.getIntParameter(request, "numPrimes", 50);
    int numDigits =
      ServletUtilities.getIntParameter(request, "numDigits", 120);
    PrimeList primeList =
      findPrimeList(primeListVector, numPrimes, numDigits);
    if (primeList == null) {
      primeList = new PrimeList(numPrimes, numDigits, true);
      synchronized(primeListVector) {
        if (primeListVector.size() >= maxPrimeLists)
          primeListVector.removeElementAt(0);
        primeListVector.addElement(primeList);
      }
    }
    Vector currentPrimes = primeList.getPrimes();
    int numCurrentPrimes = currentPrimes.size();
    int numPrimesRemaining = (numPrimes - numCurrentPrimes);
    boolean isLastResult = (numPrimesRemaining == 0);
    if (!isLastResult) {
      response.setHeader("Refresh", "5");
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Some " + numDigits + "-Digit Prime Numbers";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H2 ALIGN=CENTER>" + title + "</H2>\n" +
                "<H3>Primes found with " + numDigits +
                " or more digits: " + numCurrentPrimes +
                    ".</H3>");
    if (isLastResult)
      out.println("<B>Done searching.</B>");
    else
      out.println("<B>Still looking for " +
       numPrimesRemaining +
                  " more<BLINK>...</BLINK></B>");
```

```
   out.println("<OL>");
   for(int i=0; i<numCurrentPrimes; i++) {
     out.println("  <LI>" + currentPrimes.elementAt(i));
   }
   out.println("</OL>");
   out.println("</BODY></HTML>");
 }


 public void doPost(HttpServletRequest request,
                    HttpServletResponse response)
     throws ServletException, IOException {
   doGet(request, response);
 }
```

// See if there is an existing ongoing or completed calculation with

// the same number of primes and length of prime. If so, return

// those results instead of starting a new background thread. Keep

// this list small so that the Web server doesn't use too much memory.

// Synchronize access to the list since there may be multiple   simultaneous

// requests.

```
 private PrimeList findPrimeList(Vector primeListVector,
                                int numPrimes,
                                int numDigits) {
   synchronized(primeListVector) {
     for(int i=0; i<primeListVector.size(); i++) {
       PrimeList primes =
(PrimeList)primeListVector.elementAt(i);
       if ((numPrimes == primes.numPrimes()) &&
           (numDigits == primes.numDigits()))
         return(primes);
     }
     return(null);
   }
 }
}
```

The following Listings shows the code for PrimeList.java and Primes.java

Listing 1: Prime List. Java

```
package hall;
import java.util.*;
import java.math.BigInteger;


/** Creates a Vector of large prime numbers, usually in a low-priority background thread.
Provides a few small thread-safe access methods.*/
public class PrimeList implements Runnable {
private Vector primes;
private int numPrimes, numDigits;

// Finds numPrimes prime numbers, each of which are
// numDigits long or longer.

public PrimeList(int numPrimes, int numDigits,
boolean runInBackground) {
// Using Vector instead of ArrayList
// to support JDK 1.1 servlet engines
primes = new Vector(numPrimes);
this.numPrimes = numPrimes;
this.numDigits = numDigits;
if (runInBackground) {
Thread t = new Thread(this);
// Use low priority so you don't slow down server.
t.setPriority(Thread.MIN_PRIORITY);
t.start();
} else {
run();
}
}

public void run() {
BigInteger start = Primes.random(numDigits);
for(int i=0; i<numPrimes; i++) {
start = Primes.nextPrime(start);
synchronized(this) {
primes.addElement(start);
}
}
}

public synchronized boolean isDone() {
```

```
return(primes.size() == numPrimes);
}

public synchronized Vector getPrimes() {
if (isDone())
return(primes);
else
return((Vector)primes.clone());
}

public int numDigits() {
return(numDigits);
}
public int numPrimes() {
return(numPrimes);
}

public synchronized int numCalculatedPrimes() {
return(primes.size());
}
}
```

Listing 2: Primes. Java

package hall;

import java.math.BigInteger;

/** A few utilities to generate a large random BigInteger,

 * and find the next prime number above a given BigInteger.

 */

```
public class Primes {
  // Note that BigInteger.ZERO was new in JDK 1.2, and 1.1
  // code is being used to support the most servlet engines.
  private static final BigInteger ZERO = new   BigInteger("0");
  private static final BigInteger ONE = new         BigInteger("1");
  private static final BigInteger TWO = new         BigInteger("2");

  // Likelihood of false prime is less than 1/2^ERR_VAL
  // Assumedly BigInteger uses the Miller-Rabin test or
  // equivalent, and thus is NOT fooled by Carmichael numbers.
  // See section 33.8 of Cormen et al's Introduction to Algorithms
```

```
// for details.
private static final int ERR_VAL = 100;

public static BigInteger nextPrime(BigInteger start) {
  if (isEven(start))
    start = start.add(ONE);
  else
    start = start.add(TWO);
  if (start.isProbablePrime(ERR_VAL))
    return(start);
  else
    return(nextPrime(start));
}

private static boolean isEven(BigInteger n) {
  return(n.mod(TWO).equals(ZERO));
}

private static StringBuffer[] digits =
  { new StringBuffer("0"), new StringBuffer("1"),
    new StringBuffer("2"), new StringBuffer("3"),
    new StringBuffer("4"), new StringBuffer("5"),
    new StringBuffer("6"), new StringBuffer("7"),
    new StringBuffer("8"), new StringBuffer("9") };

private static StringBuffer randomDigit() {
  int index = (int)Math.floor(Math.random() * 10);
  return(digits[index]);
}

public static BigInteger random(int numDigits) {
  StringBuffer s = new StringBuffer("");
  for(int i=0; i<numDigits; i++) {
    s.append(randomDigit());
  }
  return(new BigInteger(s.toString()));
}

public static void main(String[] args) {
  int numDigits;
  if (args.length > 0)
    numDigits = Integer.parseInt(args[0]);
  else
```

```
      numDigits = 150;
    BigInteger start = random(150);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

PrimeNumbers.html

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Finding Large Prime Numbers</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Finding Large Prime Numbers</H2>
<BR><BR>
<CENTER>
<FORM ACTION="/servlet/hall.PrimeNumbers">
  <B>Number of primes to calculate:</B>
  <INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
  <B>Number of digits:</B>
  <INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
  <INPUT TYPE="SUBMIT" VALUE="Start Calculating">
</FORM>
</CENTER>

</BODY>
</HTML>
```
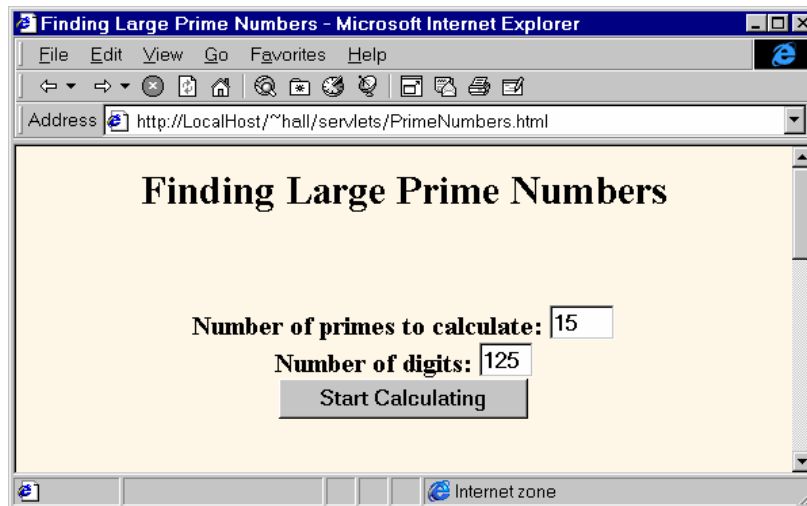
Figure 25.1 Front End
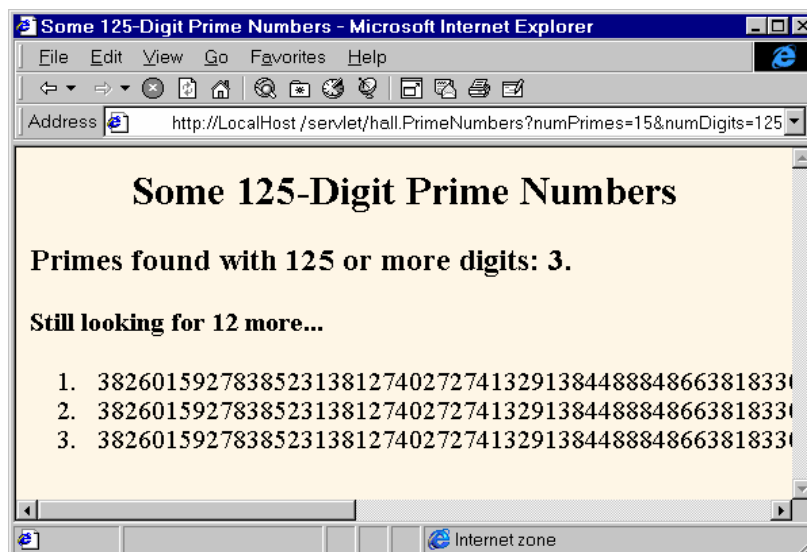
Figure 25.2 Intermediate Result
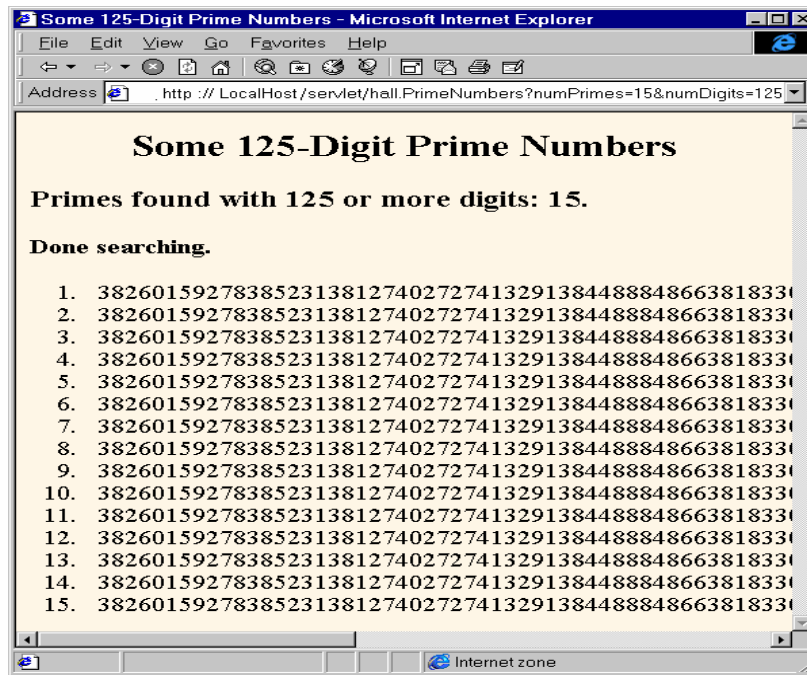
Figure 25. 3 Final Result

25.2 short summary

- Response headers can be used to specify cookies, to supply the modification date (for caching), to instruct the browser to reload the page after a designated interval….

25.3 Brain Storm

1. Write short note on Common Response Headers and their purpose.

Lecture 26

Cookies

Objectives

In this lecture you are going to learn the following

- About cookies

- About Cokkie Utilities

<div style="border:3px double black; text-align:center;">

Coverage Plan

</div>

Lecture 26

26.1 Snap Shot - Overview of Cookies

Cookies are small bits of textual information that a Web server sends to a browser and that the browser returns unchanged when visiting the same Web site or domain later. Servlets send cookies to clients by adding fields to HTTP response headers. Clients automatically return cookies by adding fields to HTTP request headers.

By having the server read information it sent the client previously, the site can provide visitors with a number of conveniences:

- **Identifying a user during an e-commerce session.** Many on-line stores use a "shopping cart" metaphor in which the user selects an item, adds it to his shopping cart, and then continues shopping. Since the HTTP connection is closed after each page is sent, when the user selects a new item for his cart, how does the store know that he is the same user that put the previous item in his cart? Cookies are a good way of accomplishing this. In fact, this is so useful that Servlets have an API specifically for this, and Servlet authors don't need to manipulate cookies directly to make use of it. This is discussed in the next Section.

- **Avoiding username and password.** Many large sites require you to register in order to use their services, but it is inconvenient to remember the username and password. Cookies are a good alternative for low-security sites. When a user registers, a cookie is sent with a unique user ID. When the client reconnects at a later date, the user ID is returned, the server looks it up, determines it belongs to a registered user, and doesn't require an explicit username and password.

- **Customizing a site.** Many "portal" sites let you customize the look of the main page. They use cookies to remember what you wanted, so that you get that result initially next time.

- **Focusing advertising.** The search engines charge their customers much more for displaying "directed" ads than "random" ads. That is, if you do a search on "Java Servlets", a search site can charge much more for an ad for a Servlet development environment than an ad for an on-line travel agent. On the other hand, if the search had been "Bali Hotels", the situation would be reversed. The problem is that they have to show a random ad when you first arrive and haven't yet performed a search, as well as when you search on

something that doesn't match any ad categories. Cookies let them remember "Oh, that's the person who was searching for such and such previously" and display an appropriate (read "high priced") ad instead of a random (read "cheap") one.

Now, providing convenience to the user and added value to the site owner is the purpose behind cookies. And despite much misinformation, cookies are not a serious security threat. Cookies are never interpreted or executed in any way, and thus can't be used to insert viruses or attack your system in any way. Furthermore, since browsers generally only accept 20 cookies per site and 300 cookies total, and each cookie is limited to 4KB, cookies cannot be used to fill up someone's disk or launch other denial of service attacks. However, even though they don't present a serious security threat, they can present a significant threat to privacy. First, some people don't like the fact that search engines can remember that they're the person that usually does searches on such and such a topic.

For example, they might search for job openings or health data, and don't want some banner ad tipping off their coworkers next time they do a search. Even worse, two search engines could share data on a user by both loading small images off a third party site, where that third party uses cookies and shares the data with both search engines. (Netscape, however, provides a nice feature that lets you refuse cookies from sites other than that to which you connected, but without disabling cookies altogether.) This trick can even be exploited via email if you use an HTML-enabled email reader that "supports" cookies, as Outlook Express does.

Thus, people could send you email that loads images, attach cookies to those images, then later identify you (email address and all) when you went to their site. Or, a site that ought to have much higher security standards might let users skip user name and passwords via cookies. For example, some of the big on-line bookstores use cookies to remember users, and let you order without reentering much of your personal information.

However, they don't actually display the full credit card number, and only let you send books to an address that was entered when you did enter the credit card in full or use the username and password. As a result, someone using the person's computer (or stealing their cookie file) could do no more harm than sending a big book order to the credit card owner's address, where it could be refused. However, smaller companies might not be so careful, and access to someone's computer or cookie file could result in loss of valuable personal information. Even

worse, incompetent sites might embed credit card or other sensitive information directly in the cookies themselves, rather than using innocuous identifiers which are only linked to real users on the server.

The point of all this is two fold. First, due to real and perceived privacy problems, some users turn off cookies. So, even when you use cookies to give added value to a site, your site shouldn't depend on them.

Second, as the author of Servlets that use cookies, you should be careful not to trust cookies for particularly sensitive information, since this would open the user up to risks if somebody accessed their computer or cookie files.

26.1.1 The Servlet Cookie API

To send cookies to the client, a servlet would create one or more cookies with the appropriate names and values via new Cookie(name, value) (section 2.1), set any desired optional attributes via cookie.setXxx (section 2.2), and add the cookies to the response headers via response.addCookie(cookie) (section 2.3). To read incoming cookies, call request.getCookies(), which returns an array of Cookie objects. In most cases, you loop down this array until you find the one whose name (getName) matches the name you have in mind, then call getValue on that Cookie to see the value associated with that name.

Creating Cookies

A Cookie is created by calling the Cookie constructor, which takes two strings : the cookie name and the cookie value. Neither the name nor the value should contain whitespace or any of: [] () =, " /? @:

Reading and Specifying Cookie Attributes

Before adding the cookie to the outgoing headers, you can look up or set attributes of the cookie. Here's a summary: getComment/setComment Gets/sets a comment associated with this cookie.

getDomain/setDomain

Gets/sets the domain to which cookie applies. Normally, cookies are returned only to the exact hostname that sent them. You can use this method to instruct the browser to return them to other hosts within the same domain. Note that the domain should start with a dot (e.g.. prenhall.com), and must contain two dots for non-country domains like .com, .edu, and .gov, and three dots for country domains like. Co.uk and. edu.es.

getMaxAge/setMaxAge

Gets/sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session (i.e. until the user quits the browser), and will not be stored on disk. See the LongLivedCookie class below, which defines a subclass of Cookie with a maximum age automatically, set one year in the future.

getName/setName

Gets/sets the name of the cookie. The name and the value are the two pieces you virtually always care about. Since the getCookies method of HttpServletRequest returns an array of Cookie objects, it is common to loop down this array until you have a particular name, then check the value with getValue. See the getCookieValue method shown below.

getPath/setPath

Gets/sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories. This method can be used to specify something more general. For example, someCookie.setPath("/") specifies that all pages on the server should receive the cookie. Note that the path specified must include the current directory.

getSecure/setSecure

Gets/sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.

getValue/setValue

Gets/sets the value associated with the cookie. Again, the name and the value are the two parts of a cookie that you almost always care about, although in a few cases a name is used as a boolean flag, and its value is ignored (i.e the existence of the name means true).

getVersion/setVersion

Gets/sets the cookie protocol version this cookie complies with. Version 0, the default, adheres to the original Netscape specification.

Placing Cookies in the Response Headers

The cookie is added to the Set-Cookie response header by means of the addCookie method of HttpServletResponse. Here's an example:
Cookie userCookie = new Cookie("user", "uid1234");
response.addCookie(userCookie);

Reading Cookies from the Client

To send cookies to the client, you created a Cookie then used addCookie to send a Set-Cookie HTTP response header. This was discussed above in section 2.1. To read the cookies that come back from the client, you call getCookies on the HttpServletRequest. This returns an array of Cookie objects corresponding to the values that came in on the Cookie HTTP request header. Once you have this array, you typically loop down it, calling getName on each Cookie until you find one matching the name you have in mind. You then call getValue on the matching Cookie, doing some processing specific to the resultant value. This is such a common process that the following section presents a simple getCookieValue method that, given the array of cookies, a name, and a default value, returns the value of the cookie matching the name, or, if there is no such cookie, the designated default value.

26.2 Some Minor Cookie Utilities

Here are some simple but useful utilities for dealing with cookies.

Getting the Value of a Cookie with a Specified Name

Here's a section of ServletUtilities.java that slightly simplifies the retrieval of a cookie value given a cookie name by looping through the array of available Cookie objects, returning the

value of any Cookie whose name matches the input. If there is no match, the designated default value is returned.

```
  public static String getCookieValue(Cookie[] cookies,
                                      String cookieName,
                                      String defaultValue) {
    for(int i=0; i<cookies.length; i++) {
      Cookie cookie = cookies[i];
      if (cookieName.equals(cookie.getName()))
        return(cookie.getValue());
    }
    return(defaultValue);
  }
```

Long Lived Cookie. Java

Here's a small class that you can use instead of Cookie if you want your cookie to automatically persist when the client quits the browser.

```
package hall;

import javax.servlet.http.*;

public class LongLivedCookie extends Cookie {
  public static final int SECONDS_PER_YEAR = 60*60*24*365;

  public LongLivedCookie(String name, String value) {
    super(name, value);
    setMaxAge(SECONDS_PER_YEAR);
  }
}
```

Example: A Customized Search Engine Interface

Here's a variation of the Search Engines example shown before. In this version, the front end is dynamically generated instead of coming from a static HTML file. Then, the Servlet that reads the parameters and forwards them to the appropriate search engine also returns cookies to the client that list these values. Next time the client visits the front end, the cookie values are used to preload the form fields with the most recently used entries.

Search Engines Front End. Java

This servlet builds the form-based front end to the search engine servlet. At first blush, the output looks just like the page given by the static HTML page. Here, however, selected values are remembered in cookies (set by the CustomizedSearchEngines servlet that this page sends data to), so if the user comes back to the same page at a later time (even after quitting the browser and restarting), the page is initialized with the values from the previous search.

Note : The code uses SevletUtilities.java for the getCookieValue method (shown above) and for headWithTitle for generating part of the HTML. It also uses the LongLivedCookie class, shown above, for creating a Cookie that automatically has a long-term expiration date.

package hall;

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

public class SearchEnginesFrontEnd extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    Cookie[] cookies = request.getCookies();
    String searchString =
      ServletUtilities.getCookieValue(cookies,
                                      "searchString",
                                      "Java Programming");
    String numResults =
      ServletUtilities.getCookieValue(cookies,
                                      "numResults",
                                      "10");
    String searchEngine =
      ServletUtilities.getCookieValue(cookies,
                                      "searchEngine",
                                      "goggle");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Searching the Web";
    out.println(ServletUtilities.headWithTitle(title) +
```

```
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1 ALIGN=\"CENTER\">Searching the
Web</H1>\n" +
                    "\n" +
                    "<FORM ACTION=\"/servlet/hall.CustomizedSearchEngines\">\n"
+
                    "<CENTER>\n" +
                    "Search String:\n" +
                    "<INPUT TYPE=\"TEXT\" NAME=\"searchString\"\n" +
                    "VALUE=\"" + searchString + "\"><BR>\n" +
                    "Results to Show Per Page:\n" +
                    "<INPUT TYPE=\"TEXT\" NAME=\"numResults\"\n" +
                    "VALUE=" + numResults + " SIZE=3><BR>\n" +
                    "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
"        VALUE=\"goggle\"" +
                    checked("goggle", searchEngine) + ">\n" +
                    "Goggle |\n" +
                    "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
                    "VALUE=\"infoseek\"" +
                    checked("infoseek", searchEngine) + ">\n" +
                    "Infoseek |\n" +
                    "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
                    "VALUE=\"lycos\"" +
                    checked("lycos", searchEngine) + ">\n" +
                    "Lycos |\n" +
                    "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
                    "VALUE=\"hotbot\"" +
                    checked("hotbot", searchEngine) + ">\n" +
                    "HotBot\n" +
                    "<BR>\n" +
                 "<INPUT TYPE=\"SUBMIT\" VALUE=\"Search\">\n" +
                    "</CENTER>\n" +
                    "</FORM>\n" +
                    "\n" +
                    "</BODY>\n" +
                    "</HTML>\n");
  }

  private String checked(String name1, String name2) {
    if (name1.equals(name2))
      return(" CHECKED");
    else
      return("");
  }
}
```

Customized Search Engines. Java

The SearchEnginesFrontEnd servlet shown above sends its data to the CustomizedSearchEngines servlet constructing a URL for a search engine and sending a redirection response to the client, the servlet also sends cookies recording the user data. These cookies will, in turn, be used by the servlet building the front end to initialize the entries in the HTML forms.

```
package hall;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
```

/** A variation of the SearchEngine servlet that uses

 * cookies to remember users choices. These values

 * are then used by the SearchEngineFrontEnd servlet

 * to create the form-based front end with these

 * choices preset.*/

```
public class CustomizedSearchEngines extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {

    String searchString = request.getParameter("searchString");
    Cookie searchStringCookie =
      new LongLivedCookie("searchString", searchString);
    response.addCookie(searchStringCookie);
    searchString = URLEncoder.encode(searchString);
    String numResults = request.getParameter("numResults");
    Cookie numResultsCookie =
      new LongLivedCookie("numResults", numResults);
    response.addCookie(numResultsCookie);
    String searchEngine = request.getParameter("searchEngine");
    Cookie searchEngineCookie =
      new LongLivedCookie("searchEngine", searchEngine);
    response.addCookie(searchEngineCookie);
    SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
    for(int i=0; i<commonSpecs.length; i++) {
```

```
      SearchSpec searchSpec = commonSpecs[i];
      if (searchSpec.getName().equals(searchEngine)) {
        String url =
          searchSpec.makeURL(searchString, numResults);
        response.sendRedirect(url);
        return;
      }
    }
    response.sendError(response.SC_NOT_FOUND,
                "No recognized search engine specified.");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Search Engines Front End Output

Here's the front end as it looks after the user types in some values or if the user comes back to the page in the same or a later session, after having typed in the values in the previous visit.
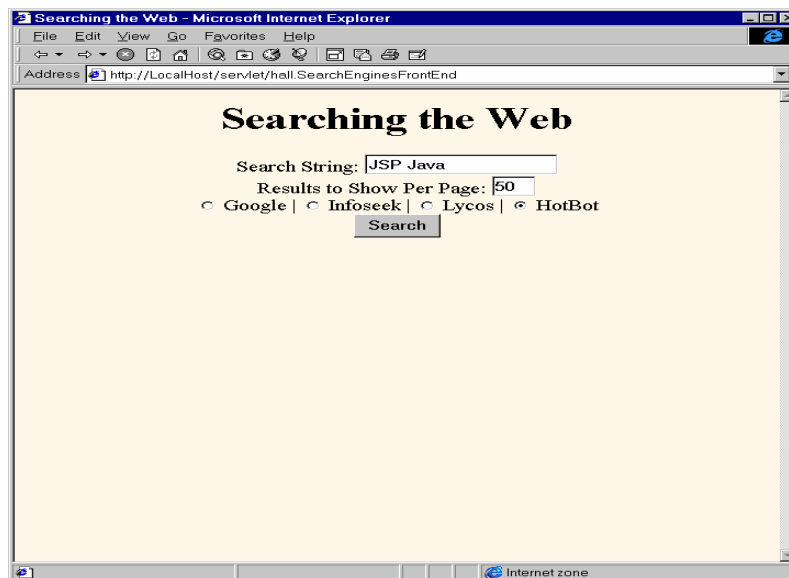
Figure 26.1 Search Engines Front End Output

26.3 Short Summary

- To send cookies to the client, a servlet would create one or more cookies with the appropriate names and values…

- A Cookie is created by calling the Cookie constructor, which takes two strings…

- Servlets send cookies to clients by adding fields to HTTP response headers.Before adding the cookie to the outgoing headers, set attributes of the cookie…

26.4 Brain Storm

1. What are cookies? Explain
2. How can you Place Cookies in the Response Headers?
3. How do you Read Cookies from the Client?

ഇ...ര

Lecture 27

Session Tracking

Objectives

In this lecture you will learn the following….

❧ Servlet communication

❧ Applet –Servlet communication

❧ Calling Servlets From Servlets (JSDK 2.0)

Coverage Plan

Lecture 27

27.1 Session tracking
27.1 Snap shot

27.1.1 The Session Tracking API

27.2 Servlet Communication
27.2.1 Applet –Servlet communication

27.2.2 Calling Servlets From Servlets (JSDK 2.0)

27.3 Short summary

27.4 Brain storm

27.1 Snap Shot

There are a number of problems that arise from the fact that HTTP is a "stateless" protocol. In particular, when you are doing on-line shopping, it is a real annoyance that the Web server can't easily remember previous transactions. This makes applications like shopping carts very problematic: when you add an entry to your cart how does the server know what's already in your cart? Even if servers did retain contextual information, you'd still have problems with e-commerce. When you move from the page where you specify what you want to buy (hosted on the regular Web server) to the page that takes your credit card number and shipping address (hosted on the secure server that uses SSL), how does the server remember what you were buying?

There are three typical solutions to this problem.

1. **Cookies.** You can use HTTP cookies to store information about a shopping session, and each subsequent connection can look up the current session and then extract information about that session from some location on the server machine. This is an excellent alternative, and is the most widely used approach. However, even though Servlets have a <u>high-level and easy-to-use interface to cookies</u> <u>high-level and easy-to-use interface to cookies</u>, there are still a number of relatively tedious details that need to be handled:

- Extracting the cookie that stores the session identifier from the other cookies (there may be many, after all),

- Setting an appropriate expiration time for the cookie (sessions interrupted by 24 hours probably should be reset), and

- Associating information on the server with the session identifier (there may be far too much information to actually store it in the cookie, plus sensitive data like credit card numbers should *never* go in cookies).

2. **URL Rewriting.** You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session. This is also an excellent solution, and even has the advantage that it works with browsers that don't support cookies or where the user has disabled cookies.

However, it has most of the same problems as cookies, namely that the server-side program has a lot of straightforward but tedious processing to do. In addition, you have to be very careful that every URL returned to the user (even via indirect means like Location fields in server redirects) has the extra information appended. And, if the user leaves the session and comes back via a bookmark or link, the session information can be lost.

3. **Hidden form fields.** HTML forms have an entry that looks like the following: <INPUT TYPE="HIDDEN" NAME="session" VALUE="...">. This means that, when the form is submitted, the specified name and value are included in the GET or POST data. This can be used to store information about the session. However, it has the major disadvantage that it only works if every page is dynamically generated, since the whole point is that each session has a unique identifier.

Servlets provide an outstanding technical solution: the HttpSession API. This is a high-level interface built on top of cookies or URL-rewriting. In fact, on many servers, they use cookies if the browser supports them, but automatically revert to URL-rewriting when cookies are unsupported or explicitly disabled. But the Servlet author doesn't need to bother with many of the details, doesn't have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store data that is associated with each session.

27.1.1 The Session Tracking API

Using sessions in Servlets is quite straightforward, and involves looking up the session object associated with the current request, creating a new session object when necessary, looking up information associated with a session, storing information in a session, and discarding completed or abandoned sessions.

Looking up the HttpSession **object associated with the current request.**

This is done by calling the getSession method of HttpServletRequest. If this returns null, you can create a new session, but this is so commonly done that there is an option to automatically create a new session if there isn't one already. Just pass true to getSession. Thus, your first step usually looks like this:

HttpSession session = request.getSession(true);

Looking up Information Associated with a Session.

HttpSession objects live on the server; they're just automatically associated with the requester by a behind-the-scenes mechanism like cookies or URL-rewriting. These session objects have a built-in data structure that let you store any number of keys and associated values. In version 2.1 and earlier of the servlet API, you use getValue("key") to look up a previously stored value.

The return type is Object, so you have to do a typecast to whatever more specific type of data was associated with that key in the session. The return value will be null if there is no such attribute. In version 2.2, getValue is deprecated in favor of getAttribute, both because of the better naming match with setAttribute (the match for getValue is putValue, not setValue), and because setAttribute lets you use an attached HttpSessionBindingListener to monitor values, while putValue doesn't. Nevertheless, since few commercial servlet engines yet support version 2.2, I'll use getValue in my examples. Here's one representative example, assuming ShoppingCart is some class you've defined yourself that stores information on items being purchased.

```
HttpSession session = request.getSession(true);
ShoppingCart previousItems =
  (ShoppingCart)session.getValue("previousItems");
if (previousItems != null) {
  doSomethingWith(previousItems);
} else {
  previousItems = new ShoppingCart(...);
  doSomethingElseWith(previousItems);
}
```

In most cases, you have a specific attribute name in mind, and want to find the value (if any) already associated with it. However, you can also discover all the attribute names in a given session by calling getValueNames, which returns a String array. In version 2.2, use getAttributeNames, which has a better name and which is more consistent in that it returns

an Enumeration, just like the getHeaders and getParameterNames methods of HttpServletRequest.

Although the data that was explicitly associated with a session is the part you care most about, there are some other pieces of information that are sometimes useful as well.

- **getId.** This method returns the unique identifier generated for each session. It is sometimes used as the key name when there is only a single value associated with a session, or when logging information about previous sessions.

- **isNew.** This returns true if the client (browser) has never seen the session, usually because it was just created rather than being referenced by an incoming client request. It returns false for preexisting sessions.

- **getCreationTime.** This returns the time, in milliseconds since the epoch, at which the session was made. To get a value useful for printing out, pass the value to the Date constructor or the setTimeInMillis method of GregorianCalendar.

- **getLastAccessedTime.** This returns the time, in milliseconds since the epoch, at which the session was last sent from the client.

- **getMaxInactiveInterval.** This returns the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never timeout.

Associating Information with a Session

As discussed in the previous section, you read information associated with a session by using getValue (or getAttribute in version 2.2 of the servlet spec). To specify information, you use putValue (or setAttribute in version 2.2), supplying a key and a value. Note that putValue replaces any previous values. Sometimes that's what you want (as with the referringPage entry in the example below), but other times you want to retrieve a previous value and augment it (as with the previousItems entry below). Here's an example:

```
HttpSession session = request.getSession(true);
```

```
  session.putValue("referringPage", request.getHeader("Referrer"));
  ShoppingCart previousItems =
    (ShoppingCart)session.getValue("previousItems");
  if (previousItems == null) {
    previousItems = new ShoppingCart(...);
  }
  String itemID = request.getParameter("itemID");
  previousItems.addEntry(Catalog.getEntry(itemID));
  // You still have to do putValue, not just modify the cart, since
  // the cart may be new and thus not already stored in the session.
session.putValue("previousItems", previousItems);
```

Example: Showing Session Information

Here is a simple example that generates a Web page showing some information about the current session.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import java.util.*;
import hall.ServletUtilities;

/** Simple example of session tracking.*/

public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Searching the Web";
    String heading;
    Integer accessCount = new Integer(0);;
    if (session.isNew()) {
      heading = "Welcome, Newcomer";
    } else {
      heading = "Welcome Back";
      Integer oldAccessCount =
        // Use getAttribute, not getValue, in version
```

```
      // 2.2 of servlet API.
      (Integer)session.getValue("accessCount");
   if (oldAccessCount != null) {
     accessCount =
       new Integer(oldAccessCount.intValue() + 1);
   }
  }
  // Use putAttribute in version 2.2 of servlet API.
  session.putValue("accessCount", accessCount);

  out.println(ServletUtilities.headWithTitle(title) +
          "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
          "<H2>Information on Your Session:</H2>\n" +
          "<TABLE BORDER=1 ALIGN=CENTER>\n" +
          "<TR BGCOLOR=\"#FFAD00\">\n" +
          "  <TH>Info Type<TH>Value\n" +
          "<TR>\n" +
          "  <TD>ID\n" +
          "  <TD>" + session.getId() + "\n" +
          "<TR>\n" +
          "  <TD>Creation Time\n" +
          "  <TD>" + new Date(session.getCreationTime()) + "\n" +
          "<TR>\n" +
          "  <TD>Time of Last Access\n" +
          "  <TD>" + new Date(session.getLastAccessedTime()) + "\n" +
          "<TR>\n" +
          "  <TD>Number of Previous Accesses\n" +
          "  <TD>" + accessCount + "\n" +
          "</TABLE>\n" +
          "</BODY></HTML>");
 }
 public void doPost(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  doGet(request, response);
 }
}
```

Here's a typical result, shown after visiting the page several without quitting the browser in between:
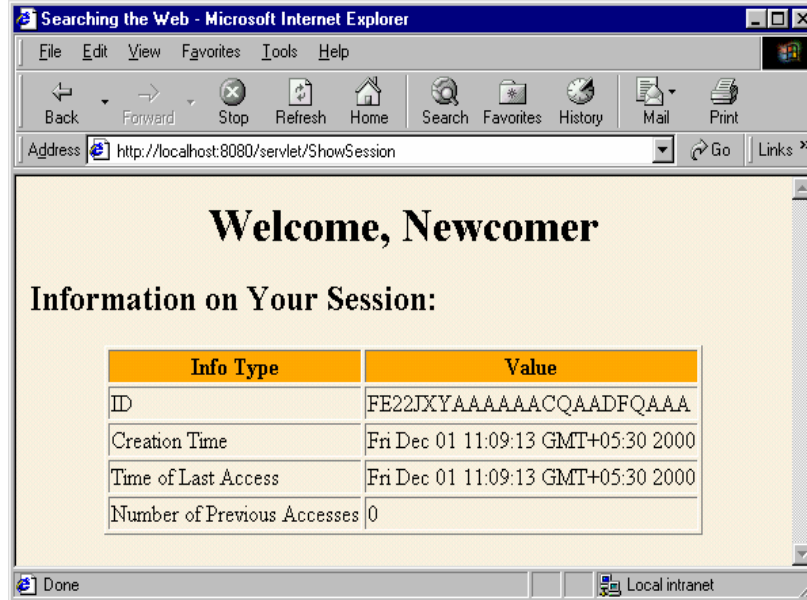
Figure 27.1 Session Information

27.2 Servlet Communication

27.2.1 Applet –Servlet communication

The applet and servlet developed in this example follow a very simple communication scheme. The applet sends some data to the servlet, the servlet reads that data, and then echoes it back to the applet. The applet then outputs the data. Our goal is to show the communication channel and focus less on the actual data that is being passed back and forth.

The following Listing is the servlet used in this example.

Listing Test Servlet.java

```
import javax.servlet.http.*;
import java.io.*;
import javax.servlet.*;
import java.util.*;
public class testServlet extends HttpServlet {
```

Centre for Information Technology and Engineering, Manonaniam Sundaranar University

```
public void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException,
                                    java.io.IOException { DataInputStream
inData = new DataInputStream(req.getInputStream());
resp.setContentType("application/octet-stream");
ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
DataOutputStream outData = new DataOutputStream(byteOut);
byte byteVal = inData.readByte();
char charVal = inData.readChar();
int intVal = inData.readInt();
String stringVal = inData.readUTF();
outData.writeByte(byteVal);
outData.writeChar(charVal);
outData.writeInt(intVal);
outData.writeUTF(stringVal);
outData.flush();
byte[] buf = byteOut.toByteArray();
resp.setContentLength(buf.length);
 ServletOutputStream servletOut = resp.getOutputStream();
 servletOut.write(buf); servletOut.close();
}

        }
```

In the real world, most of the time, the server application is already written and your challenge is to have the applet communicate to it. This requires you to thoroughly understand the server-side program, the parameters (HTML form elements, etc.) that it requires and the specific URL (with parameters) that must be used to invoke it.

The servlet implements only one method and that is the service method which is invoked to handle requests made to the servlet. inData is the instance of DataInputStream used to read data sent by the applet. We use the methods readByte(), readChar(), readInt(), and readUTF() to read some values from the applet. Note that the order in which these methods are invoked is important and must correspond with how the applet sends in the data. This also gives you a glimpse of the flexibility on what type of data can be sent back and forth. For example, readUTF() can be used to read an entire XML document, which can then be parsed by the servlet and manipulated further.

Rather than sending the output directly to the applet, our servlet first writes the data into an array of bytes. This buffer would allow the servlet to calculate the length of the array and set the amount of data being sent back via the response object.

As you can see, the servlet is very simple. This is in part due to the fact that no data manipulation is done by the servlet. A more practical example would be where the servlet invokes JDBC, RMI, EJB or CORBA methods to place the data it has received into a database or perform some complex calculations based on the data. In fact, a generic implementation of this type of solution would make the servlet completely data-independent and use it as a proxy to some other process. This way, the servlet is only used as a pass-through element in the architecture, but nonetheless important, since without it, the applet could not make direct communication with the back-end processes.

The applet must create a DataOutputStream, write some data to it, then wait for a response from the servlet. Again, our applet is very simple. All it does, is echo the data sent back to it. To see the data, use the Java Console feature on your browser, otherwise you will not see any of the applet's output. Listing 2 is the code for the applet.

Listing1 : testApplet.java

```java
import java.io.*;
import java.awt.*;
import java.applet.*;
public class testApplet extends Applet {

   public void init() {
        setLayout(null);
         setSize(426,266);
         goButton.setLabel("GO");
         add(goButton);
         goButton.setBackground(java.awt.Color.lightGray);
         goButton.setBounds(144,12,101,39);
         SymMouse aSymMouse = new SymMouse();
         goButton.addMouseListener(aSymMouse);
    }

    java.awt.Button goButton = new java.awt.Button();
   class SymMouse extends java.awt.event.MouseAdapter {
    public void mouseClicked(java.awt.event.MouseEvent event) {
          Object object = event.getSource();

      }
```

```
 }

void  goButton_MouseClicked(java.awt.event.MouseEvent event) {
try {
        System.out.println("Attempting to connect to

http://localhost:8080/examples/servlet/testServlet");
          java.net.URL url = new
java.net.URL("http://localhost:8080/examples/servlet/testServlet");
          java.net.URLConnection c = url.openConnection();
c.setUseCaches(false);
c.setDoOutput(true);
c.setDoInput(true);
ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
DataOutputStream outData = new DataOutputStream(byteOut);
System.out.println("Writing test data");
outData.writeByte(1);
outData.writeChar(2);
outData.writeInt(3);
outData.writeUTF("Test Message");
outData.flush(); byte buf[] = byteOut.toByteArray();
c.setRequestProperty("Content-type", "application/octet-stream");
c.setRequestProperty("Content-length", "" + buf.length);
DataOutputStream dataOut = new DataOutputStream(c.getOutputStream());
dataOut.write(buf);
dataOut.flush();
dataOut.close();
System.out.println("Reading response");
DataInputStream inData = new DataInputStream(c.getInputStream());
byte byteVal = inData.readByte();
char charVal = inData.readChar();
 int intVal = inData.readInt();
String stringVal = inData.readUTF();
inData.close();
System.out.println("Data read: " + byteVal + " " + ((int) charVal) + " " +
intVal + " " + stringVal);
} catch (Exception e) {
    e.printStackTrace();
}

 }
}
```

Listing 2 : testApplet.java

The applet is nothing more than a button. When you click on the button, that's when the communication occurs, so the code you need to focus on is in the event handler method goButton_MouseClicked().

Here the the URL is hardcode to the servlet. Most likely you want to get the URL as a parameter to the applet. Once again, instead of writing the data out directly, we first write it to a buffer, because we need the length of the data before sending it out.

Here we use the Java Server Web Development Kit (JSWDK) from Sun to deploy the servlet and the HTML page containing the applet. Here is the HTML page that is been used:

```
<HTML>
<HEAD>
<TITLE>testing
Applet-Servlet Communication</TITLE>
</HEAD>
<BODY>
<APPLET
CODE="testApplet.class" WIDTH=426 HEIGHT=266></APPLET>
</BODY>
</HTML>
```

After you compile the applet and the servlet you are ready to deploy. You should place the HTML page and the two classes generated by the testApplet.java code into the webpages subdirectory under JSWDK. The servlet (testServlet.class) should be placed under the examples\WEB-INF\servlets subdirectory under JSWDK. You can now start the servlet engine. In case of JSWDK, there is a batch file that starts the engine.

Using your browser, open up the HTML page. Note that you should get the page using HTTP and not load it locally (i.e., via file:///). If you use JSWDK, the URL would be something like http://localhost:8080/test.html, but you can use any Web server and adjust the URL accordingly. Once the applet is loaded, start the Java console from your browser. In Netscape, it is under Communicator - Tools.

You can now, click on the button on the applet and see the following message on the Java console indicating that the applet connected to the servlet, sent it some data, and received some data back.

27.2.2 Calling Servlets From Servlets (JSDK 2.0)

To have your servlet call another servlet, you can either:

- A servlet can make an HTTP request of another servlet. Opening a connection to a URL is discussed in the <u>Working with URLs</u> lesson
- A servlet can call another servlet's public methods directly, if the two servlets run within the same server.

  This section addresses the second option. To call another servlet's public methods directly, you must:

- Know the name of the servlet that you want to call.
- Gain access to that servlet's Servlet object
- Call the servlet's public method

To gain access to the Servlet object, use the ServletContext object's getServlet method. Get the ServletContext object from the ServletConfig object stored in the Servlet object. An example should make this clear. When the BookDetail servlet calls the BookDB servlet, the BookDetail servlet obtains the BookDB servlet's Servlet object like this:

```
public class BookDetailServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
            BookDBServlet database = (BookDBServlet)
                getServletConfig().getServletContext().getServlet("bookdb");
        ...
    }
}
```

Once you have the servlet object, you can call any of that servlet's public methods. For example, the BookDetail servlet calls the BookDB servlet's getBookDetails method:

```
public class BookDetailServlet extends HttpServlet {
```

```
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
       throws ServletException, IOException
  {
    ...
        BookDBServlet database = (BookDBServlet)
            getServletConfig().getServletContext().getServlet("bookdb");
        BookDetails bd = database.getBookDetails(bookId);
      ...
  }
}
```

> **Note:** If the servlet that you want to call implements the single threaded model interface, your call could violate the called servlet's single threaded nature. (The server has no way to intervene and make sure your call happens when the servlet is not interacting with another client.) In this case, your servlet should make an HTTP request of the other servlet instead of calling the other servlet's methods directly.

27.3 Short Summary

- HttpSession API is a high-level interface built on top of cookies or URL-rewriting.

- getId method returns the unique identifier generated for each session….

- isNew returns true if the client (browser) has never seen the session…

- getCreationTime and getLastAccessedTime returns the time, in milliseconds

- getMaxInactiveInterval returns the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never timeout.

27.4 Brain Storm

1. Write short note on Applet –Servlet communication.
2. Write short note on Calling Servlets From Servlets (JSDK 2.0)

సౌఆ

Lecture 28

Working with URL'S

Objectives

In this lecture you will learn about

> ✍ Servlet Communication

Coverage Plan

Lecture 28

28.1 Snap Shot

This chapter is going to discuss about Servlet communication, working with URL's and etc.

28.1.1 servlet communication - Working with URLs

Parsing a URL

The URL class provides several methods that let you query URL objects. You can get the protocol, host name, port number, and filename from a URL using these accessor methods:

getProtocol
Returns the protocol identifier component of the URL.

getHost
Returns the host name component of the URL.

getPort
Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.

getFile
Returns the filename component of the URL.

getRef

Returns the reference component of the URL.

> **Note:** Remember that not all URL addresses contain these components. The URL class provides these methods because HTTP URLs do contain these components and are perhaps the most commonly used URLs. The URL class is somewhat HTTP-centric.

You can use these getXXX methods to get information about the URL regardless of the constructor that you used to create the URL object.

The URL class, along with these accessor methods, frees you from ever having to parse URLs again! Given any string specification of a URL, just create a new URL object and call any of the accessor methods for the information you need. This small example program creates a URL from a string specification and then uses the URL object's accessor methods to parse the URL:

```
import java.net.*;
import java.io.*;

public class ParseURL {
    public static void main(String[] args) throws Exception {
        URL aURL = new URL("http://java.sun.com:80/docs/books/"
                            + "tutorial/index.html#DOWNLOADING");
        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("host = " + aURL.getHost());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("port = " + aURL.getPort());
        System.out.println("ref = " + aURL.getRef());
    }
}


Here's the output displayed by the program:

protocol = http
host = java.sun.com
filename = /docs/books/tutorial/index.html
port = 80
ref = DOWNLOADING
```

28.1.2 Reading Directly from a URL

After you've successfully created a URL, you can call the URL's openStream() method to get a stream from which you can read the contents of the URL. The openStream() method returns a

java.io.InputStreamhttp://java.sun.com/products/jdk/1.2/docs/api/java.io.InputStream.html

http://java.sun.com/products/jdk/1.2/docs/api/java.io.InputStream.htmlobject, so reading from a URL is as easy as reading from an input stream.

The following small Java program uses openStream() to get an input stream on the URL http://www.yahoo.com/. It then opens a BufferedReader on the input stream and reads from the BufferedReader thereby reading from the URL. Everything read is copied to the standard output stream:

```
import java.net.*;
import java.io.*;
public class URLReader {
    public static void main(String[] args) throws Exception {
       URL yahoo = new URL("http://www.yahoo.com/");
       BufferedReader in = new BufferedReader(
                         new InputStreamReader(
                         yahoo.openStream()));
       String inputLine;
       while ((inputLine = in.readLine()) != null)
           System.out.println(inputLine);
       in.close();
    }
}
```

When you run the program, you should see, scrolling by in your command window, the HTML commands and textual content from the HTML file located at http://www.yahoo.com/. Alternatively, the program might hang or you might see an exception stack trace. If either of the latter two events occurs, you may have to set the proxy host so that the program can find the Yahoo server.

28.1.3 Connecting to a URL

After you've successfully created a URL object, you can call the URL object's openConnection method to connect to it. When you connect to a URL, you are initializing a communication link between your Java program and the URL over the network. For example, you can open a connection to the Yahoo site with the following code:

```
try {
    URL yahoo = new URL("http://www.yahoo.com/");
    URLConnection yahooConnection = yahoo.openConnection();

} catch (MalformedURLException e) {     // new URL() failed
    . . .
} catch (IOException e) {        // openConnection() failed
```

```
        . . .
    }
```

If possible, the openConnection method creates a new URLConnection (if an appropriate one does not already exist), initializes it, connects to the URL, and returns the URLConnection object. If something goes wrong--for example, the Yahoo server is down--then the openConnection method throws an IOException. Now that you've successfully connected to your URL, you can use the URLConnection object to perform actions such as reading from or writing to the connection. The next section shows you how.

28.1.4 Reading from and Writing to a URLConnection

If you've successfully used openConnection to initiate communications with a URL, then you have a reference to a URLConnection object. The URLConnection class contains many methods that let you communicate with the URL over the network. URLConnection is an HTTP-centric class; that is, many of its methods are useful only when you are working with HTTP URLs. However, most URL protocols allow you to read from and write to the connection. This section describes both functions.

28.1.5 Reading from a URLConnection

The following program performs the same function as the URLReader program shown in Reading directly from the URL. However, rather than getting an input stream directly from the URL, this program explicitly opens a connection to a URL and gets an input stream from the connection. Then, like URLReader, this program creates a BufferedReader on the input stream and reads from it. The bold statements highlight the differences between this example and the previous

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                yc.getInputStream()));
```

```
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

The output from this program is identical to the output from the program that opens a stream directly from the URL. You can use either way to read from a URL. However, reading from a URLConnection instead of reading directly from a URL might be more useful. This is because you can use the URLConnection object for other tasks (like writing to the URL) at the same time. Again, if the program hangs or you see an error message, you may have to set the proxy host so that the program can find the Yahoo server.

28.1.6 Writing to a URLConnection

Many HTML pages contain forms-- text fields and other GUI objects that let you enter data to send to the server. After you type in the required information and initiate the query by clicking a button, your Web browser writes the data to the URL over the network. At the other end, a cgi-bin script (usually) on the server receives the data, processes it, and then sends you a response, usually in the form of a new HTML page.

Many cgi-bin scripts use the POST METHOD for reading the data from the client. Thus writing to a URL is often called posting to a URL. Server-side scripts use the POST METHOD to read from their standard input.

28.2 Short Summary

- URLConnection object is to perform actions such as reading from or writing to the connection....
- After created a URL object, the URL object's openConnection method is used to connect to it for servlet communication.
- The URLConnection class contains many methods that let you communicate with the URL over the network.
- Some server side cgi-bin scripts use the GET METHOD to read the data…
- The post method is quickly making the get method obsolete because it's more versatile and has no limitations on the amount of data that can be sent through the connection

28.3 Brain Storm

1. How to the protocol, host name, port number, and filename from a URL?

2. How to read from URL?

3. How to write to URL?

4. What for the following CGI variable used for?

    - SCRIPT_NAME

    - SERVER_PORT

    - REMOTE_ADDR

    - REQUEST_METHODS

ಐಇ

Lecture 29

JSP - Introduction

Objectives

In this lecture you will learn the following:

🕮 Introduction to JSP Basics

🕮 ABOUT JSP Request Model

29.1 Snap Shot

This chapter provides an overview of JSP Request model, JSP Architecture, JSP scriptlets and JSP Actions.

29.2 JSP BASICS

JavaServerPages also known as JSPs, are one of the most powerful and simplest way of generating dynamic HTML on the server side. JSP is a presentation layer technology that sits on top of a Java Servlets model and makes working with HTML easier. Like Server Side Java Script(SSJS), it allows you to mix static HTML content with server-side scripting to produce dynamic output. By default, JSP uses Java as it's scripting language; however, the specification allows other languages to be used, just as ASP can use other languages (such as JavaScript and VBScript). While JSP with Java will be more flexible and robust than scripting platforms based on simpler languages like JavaScript and VBScript, Java also has a steeper learning curve than simple scripting languages.

To offer the best of both worlds - a robust web application platform and a simple, easy-to-use language and tool set - JSP provides a number of server-side tags that allow developers to perform most dynamic content operations without ever writing a single line of Java code. So developers who are only familiar with scripting, or even those who are simply HTML designers, can use JSP tags for generating simple output without having to learn Java. Advanced scriptures or Java developers can also use the tags, or they can use the full Java language if they want to perform advanced operations in JSP pages.

29.2.1 The Magic of JSP

"JSP is component-centric". To understand how JSP can accomplish the magic act combined with "unlimited" power, one must first understand the difference between component-centric and page-centric web development. Both SSJS and ASP were designed years ago when the web was young and no one knew any better than to dump all their business, data, and presentation logic into scripted web pages.

This page-centric model was easy to learn and allowed for fairly rapid development. However, over time people realized that this wasn't the way to build large, scalable web

applications. The logic written for the scripted environments was locked inside pages and was reusable only through cut and paste. Presentation logic was regularly mixed with business and data logic, making application maintenance an exercise in walking on eggshells, as programmers attempted to modify the look and feel of an application without breaking the tightly coupled business logic. Sophisticated, reusable components already existed in the enterprise, and no one wanted to rewrite that logic for their web applications.

HTML and graphics designers handed over the implementation of their designs to web scriptures, who had to duplicate the work - often by hand, because no decent tools existed for combining server-side scripting with HTML content generation. In short, web application complexity increased steadily and the limitations of the page-centric model became obvious. Around the same time that people were looking around for better ways to build web applications, components were all the rage in the client-server world. JavaBeans and ActiveX were being pushed to Java and Window75s application developers by rapid application development (RAD) tools vendors as the way to develop complex applications quickly. These technologies also enabled domain experts to write components for vertical applications, which developers could then use without having significant domain expertise.

Component to obtain the relevant domain-specific information, and focus instead on making the application easy to use. It was only a matter of time before a technology emerged to bring component-centric programming to the world of server-side Web applications. JSP comes out of the gate as a component-centric platform. It's based on a model in which JavaBeans and Enterprise JavaBeans (EJB) components contain the business and data logic for an application, and it provides tags and a scripting platform for exposing the content generated or returned by the beans in HTML pages. Because of the component-centric nature of JSP, it can be used by non-Java and Java developers alike. Non-Java developers can use the JSP tags to work with beans that experienced Java developers created. Java developers can not only make and use beans but also use Java in JSP pages for finer-grained control over presentation logic that's based on underlying beans.

An example

The following code shows how a simple jsp file will look like:

```
<HTML>
<BODY>
<% out.println("Hello World"); %>
</BODY>
```

```
</HTML>
```

The above program should be stored in a file with .jsp as extension (say Helloworld.jsp). From the above document, we can see that the jsp document looks like a HTML document with some added tags to support Java codings. This .jsp file should be stored in the document directory of the Web server. Soon we'll see how to run the above coding to get the desired output.

29.3 Advantages of JSP

- Vs. Active Server Pages (ASP). ASP is a similar technology from Microsoft. The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS-specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and Microsoft Web servers.

- Vs. Pure Servlets. JSP doesn't give you anything that you couldn't in principle do with a Servlet. But it is more convenient to write (and to modify!) regular HTML than to have a zillion println statements that generate the HTML. Plus, by separating the look from the content you can put different people on different tasks: your Web page design experts can build the HTML, leaving places for your Servlet programmers to insert the dynamic content.
- Vs. Server-Side Includes (SSI). SSI is a widely supported technology for including externally defined pieces into a static Web page. JSP is better because it lets you use Servlets instead of a separate program to generate that dynamic part. Besides, SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

- Vs. JavaScript. JavaScript can generate HTML dynamically on the client. This is a useful capability, but only handles situations where the dynamic information is based on the client's environment. With the exception of cookies, HTTP and form submission data is not available to JavaScript. And, since it runs on the client, JavaScript can't access server-side resources like databases, catalogs, pricing information, and the like.

- Vs. Static HTML. Regular HTML, of course, cannot contain dynamic information. JSP is so easy and convenient that it is quite feasible to augment HTML pages that only benefit

marginally by the insertion of small amounts of dynamic data. Previously, the cost of using dynamic data would preclude its use in all but the most valuable instances.

29.4 JSP Request Model

Now let's take a look at how HTTP requests are processed under the JSP model. In the basic request model, a request is sent directly to a JSP page. Figure 8.1 illustrates the flow of information in this model. JSP code controls interactions with JavaBeans components for business and data logic processing, and then displays the results in dynamically generated HTML mixed with static HTML code.
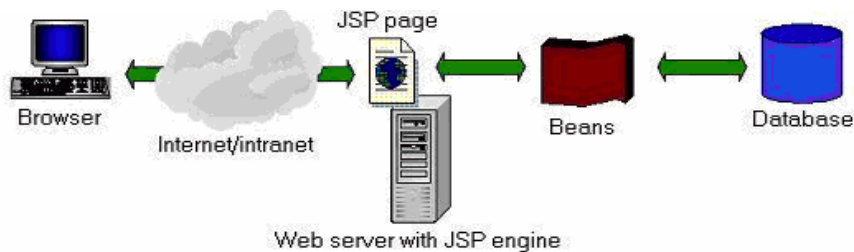
Figure 29.1 Basic JSP request model

The beans depicted can be JavaBeans or EJB components. Other, more complicated request models include calling out to other JSP pages or Java Servlets from the requested JSP page.

29.5 Short Summary

- JSP is a presentation layer technology that sits on top of a Java Servlets model and makes working with HTML easier

- Advanced scriptures or Java developers can also use the tags, or they can use the full Java language if they want to perform advanced operations in JSP pages.

29.6 Brain Storm

1. Write short notes on JSP.
2. What are the advantages of JSP?
3. Explain JSP request model.

ಬಿಲ

Lecture 30

JSP Architecture

Objectives

In this lecture you will learn the following

✍ About JSP Architecture

✍ About JSP Scripting elements

Coverage Plan

Lecture 30

30.1 Snap Shot

This lecture will discuss about JSP architecture, JSP scriplets, JSP declarations and JSP Directives.

30.2 JSP Architecture

The source code of a JSP page is essentially just HTML (or text—or even XML) sprinkled here and there with either special JSP tags and/or Java code enclosed in these tags. The file's extension is .jsp rather than the usual. html or .htm, and it tells the server that this document requires special handling.

The special handling, accomplished with a Web server extension or plug-in, involves four steps (see Figures 8.1):

1. The JSP engine parses the page and creates a Java source file.
2. It then compiles the file produced in Step 1 into a Java class file. The class file created in Step 2 is a servlet, and from this point on, the servlet engine handles the class file in the same manner as all other servlets.
3. The servlet engine loads the servlet class for execution.
4. The servlet executes and streams back the results to the requestor.

Although this process might seem time consuming and expensive, it's much more efficient than it sounds. Steps 1 and 2 occur only once, when you first deploy or update the JSP.
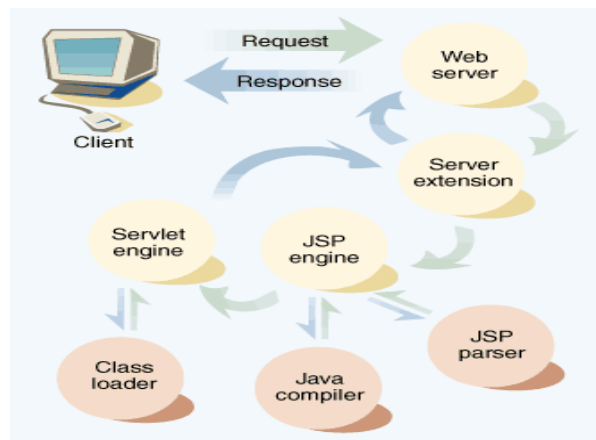

Figure 30.1 JSP Architecture

(By the way, most engines allow you to update a page on the fly.) The servlet engine performs Step 3 only upon the first request of that servlet since the last server restart. After that, the class loader loads the class once and is available for the life of that JVM. Finally, some application servers provide page caching, which can further improve the performance and reduce the cost of executing the request. With page caching, even Step 4 may execute only once depending on how dynamic the page data is.

30.3 Getting on with JSP

As said earlier Java Server Pages (JSP) lets you separate the dynamic part of your pages from the static HTML. Let us start examining the same HelloWorld.jsp example again.

```
<HTML>
<BODY>
<% out.println("Hello World"); %>
</BODY>
</HTML>
```

Listing:HelloWorld.jsp

Here the regular HTML is written in the normal manner, using whatever Web-page-building tools you normally use. Then the code for the dynamic parts alone are encoded in special tags starting with "<%" and ending with "%>".

To run the above example,

- Place the HelloWorld.jsp in the examples directory of Web server.
  (root_directory\JavaWebServer2.0\examples)

- Run the JavaWebServer.
  Go to root_directory\JavaWebServer2.0\bin
  Type httpd.

- Now open the browser and type http://localhost:8080/examples/HelloWorld.jsp in the
  place where address of the file is required.

The output might look like this
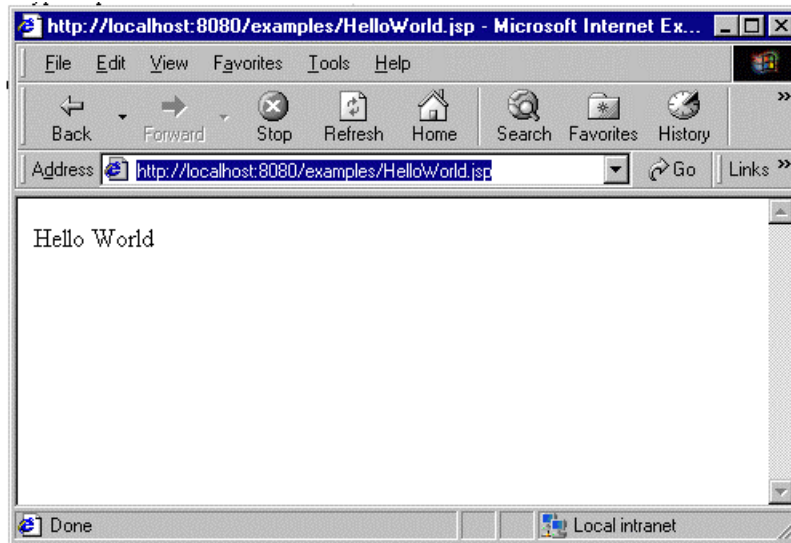
Behind the scenes:

You normally give your file a jsp extension, and typically install it in any place you could place a normal Web page. Although what you write often looks more like a regular HTML file than a Servlet, behind the scenes, the JSP page just gets converted to a normal Servlet, with the static HTML simply being printed to the output stream associated with the servlet's service method. This is normally done the first time the page is requested, and developers can simply request the page themselves when first installing it if they want to be sure that the first real user doesn't get a momentary delay when the JSP page is translated to a Servlet and the Servlet is compiled and loaded. Note also that many Web servers let you define aliases so that a URL that appears to reference an HTML file really points to a Servlet or JSP page.

30.4 Components of a JavaServerPage

Aside from the regular HTML, there are three main types of JSP constructs that you embed in a page:
Scripting elements,

Directives, and Actions.

Scripting elements let you specify Java code that will become part of the resultant Servlet, Directives let you control the overall structure of the Servlet, and Actions let you specify

existing components that should be used, and otherwise control the behavior of the JSP engine.

Template Text: Static HTML

In many cases, a large percent of your JSP page just consists of static HTML, known as template text. In all respects except one, this HTML looks just like normal HTML, follows all the same syntax rules, and is simply "passed through" to the client by the Servlet created to handle the page. Not only does the HTML look normal, it can be created by whatever tools you already are using for building Web pages. The one minor exception to the "template text is passed straight through" rule is that, if you want to have "<%" in the output, you need to put "<\ %" in the template text.

30.5 JSP Scripting Elements

JSP scripting elements let you insert Java code into the Servlet that will be generated from the current JSP page. There are three forms:

1. Expressions of the form <%= expression %> that are evaluated and inserted into the output,
2. Script lets of the form <% code %> that are inserted into the Servlets service method, and
3. Declarations of the form <%! code %> that are inserted into the body of the Servlet class, outside of any existing methods.

Each of these is described in more detail below.

30.5.1 JSP Expressions

A JSP expression is used to insert Java values directly into the output. It has the following form:

<%= Java Expression %>.

The Java expression is evaluated, converted to a string, and inserted in the page. This evaluation is performed at run-time (when the page is requested), and thus has full access to

information about the request. For example, the following listing shows the date/time that the page was requested:

Example

```
<HTML>

<HEAD>
    <TITLE>JSP DATE EXAMPLE</TITLE>
</HEAD>

<BODY>
<BIG>
 <H1>DATE</H1>
 <H2><%=new java.util.Date() %></H2>
</BIG>
</BODY>

</HTML>
```

Listing Expression.jsp

30.5.2 JSP Scriptlets

If you want to do something more complex than insert a simple expression, JSP scriptlets let you insert arbitrary code into the servlet method that will be built to generate the page. Scriptlets have the following form:

<% Java Code %>

Scriptlets have access to the same automatically defined variables as expressions. So, for example, if you want output to appear in the resultant page, you would use the out variable. The following example listing shows how to attach java coding inside a JSP page.

Example
```
<HTML>
<BODY>
<%
String queryData = request.getQueryString();
```

```
if (queryData.equals("hello"))
  out.println(queryData+" U are most welcome to JSP");
else
  out.println("Attached GET data is "+queryData);
%>
</BODY>

</HTML>
```

Listing Scriptlets.jsp

To run the above example, type

 http://localhost:8080/examples/Scriptlets.jsp?hello in the browser URL.

Note that code inside a scriptlet gets inserted exactly as written, and any static HTML (template text) before or after a scriptlet gets converted to print statements. This means that scriptlets need not contain complete Java statements, and blocks left open can affect the static HTML outside of the scriptlets. For example, the following JSP fragment, containing mixed template text and scriptlets

```
<% if (Math.random() < 0.5) { %>
Have a <B>nice</B> day!
<% } else { %>
Have a <B>lousy</B> day!
<% } %>
```

will get converted to something like:

```
if (Math.random() < 0.5) {
  out.println("Have a <B>nice</B> day!");
} else {
  out.println("Have a <B>lousy</B> day!");
}
```

Note: If you want to use the characters "%>" inside a scriptlet, enter "%\>" instead.

30.5.3 JSP Declarations

A JSP declaration lets you define methods or fields that get inserted into the main body of the servlet class (outside of the service method processing the request). It has the following form:

```
<%! Java Code %>
```

Since declarations do not generate any output, they are normally used in conjunction with JSP expressions or scriptlets. For example, here is a JSP fragment that prints out the number of times the current page has been requested since the server booted (or the Servlet class was changed and reloaded):

Example

```
<HTML>

<BODY>
<%! private int accessCount = 0; %>
ACCESSES TO THE PAGE SINCE SERVER REBOOT:
<%= ++accessCount %>
</BODY>

</HTM>L
```

Listing Declarations.jsp

Note: As with scriptlets, if you want to use the characters "%>", enter "%\>" instead.

30.5.4 JSP Directives

A JSP directive affects the overall structure of the Servlet class. It usually has the following form:

<%@ directive attribute="value" %>

However, you can also combine multiple attribute settings for a single directive, as follows:

<%@ directive attribute1="value1"

attribute2="value2"

...

attributeN="valueN" **%>**

There are two main types of directive:

- page, which lets you do things like import classes, customize the Servlet superclass, and the like; and

- include, which lets you insert a file into the Servlet class at the time the JSP file is translated into a Servlet.

The specification also mentions the taglib directive, which is not supported in JSP version 1.0, but is intended to let JSP authors define their own tags. It is expected that this will be the main new contribution of JSP 1.1.

30.5.6 The JSP Page Directive

The page directive lets you define one or more of the following case-sensitive attributes:

- import="package.class" or import="package.class1,...,package.classN". This let you specify what packages should be imported. For example:
  <%@ page import="java.util.*" %>
  The import attribute is the only one that is allowed to appear multiple times.

- contentType="MIME-Type" or
  contentType="MIME-Type; charset=Character-Set"

  This specifies the MIME type of the output. The default is text/html. For example, the directive

  <%@ page contentType="text/plain" %>
  has the same effect as the scriptlet

  <% response.setContentType("text/plain"); %>

- isThreadSafe="true|false". A value of true (the default) indicates normal Servlet processing, where multiple requests can be processed simultaneously with a single Servlet instance, under the assumption that the author synchronized access to instance variables. A value of false indicates that the Servlet should implement SingleThreadModel, with requests either delivered serially or with simultaneous requests being given separate Servlet instances.

- session="true|false". A value of true (the default) indicates that the predefined variable session (of type HttpSession) should be bound to the existing session if one exists, otherwise a new session should be created and bound to it. A value of false indicates that no sessions will be used, and attempts to access the variable session will result in errors at the time the JSP page is translated into a servlet.

- buffer="sizekb|none". This specifies the buffer size for the JspWriter out. The default is server-specific, but must be at least 8kb.

- autoflush="true|false". A value of true, the default, indicates that the buffer should be flushed when it is full. A value of false, rarely used, indicates that an exception should be thrown when the buffer overflows. A value of false is illegal when also using buffer="none".

- extends="package.class". This indicates the superclass of Servlet that will be generated. Use this with extreme caution, since the server may be using a custom superclass already.

- info="message". This defines a string that can be retrieved via the getServletInfo method.

- errorPage="url". This specifies a JSP page that should process any Throwables thrown but not caught in the current page.

- isErrorPage="true|false". This indicates whether or not the current page can act as the error page for another JSP page. The default is false.

- language="java". At some point, this is intended to specify the underlying language being used. For now, don't bother with this since java is both the default and the only legal choice.

30.5 Short Summary

- Syntax of the JSP expression is   **<%= Java Expression %>.**
- A JSP declaration lets you define methods or fields that get inserted into the main body of the servlet class.
- There are two main types of JSP directive they are page and include…
- In JSP Components

- ❖ Scripting elements let you specify Java code that will become part of the resultant Servlet,
- ❖ Directives let you control the overall structure of the Servlet,
- ❖ Actions let you specify existing components that should be used, and otherwise control the behavior of the JSP engine.

30.6 Brain Storm

1. Write short notes on JavaServerPage's components.
2. List down the JSP scripting elements.
3. What are JSP page directives?
4. What is a JSP Expression?
5. What are JSP Scriptlets and Directions?

ೞೞ

Lecture 31

Handling JSP Errors

Objectives

In this lecture you will learn the following

✎ How to handle JSP error…

✎ Predefined variables…

<div style="border:2px solid black; padding:8px;">

Coverage Plan

</div>

Lecture 31

31.1 Snap Shot - Handling JSP Errors

JSP errors are of two type: *Translation time Errors* and *Request Time Errors*. The first type of JSP error occurs when a JavaServerPage is first requested and goes through the initial translation from JSP source file into Servlet class file.Since these errors are usually compilation errors they are called as Translation time errors. They are reported to the requesting client with an error status code 500 or ServerError. These translation errors are handled by JSP engine.The second type of JSP errors occur during run-time and occurs in the body of the JSP page or in some other object called from the JSP oage.

31.2 JSP Error Page

Request time errors cause exception being thrown. If these exceptions are not handled then we can forward an error page (usually containing the description of the error) to the client.

Creating a JSP Error Page

To make a normal JSP page as an Error page, just tell the JSP engine the speciality of the page by setting page attribute isErrorPage to true.

```
<HTML>
<BODY>
<%@  page isErrorPage="true" %>
<!-- use the implicit exception object to get -->
<!-- the details about the thrown exception   -->
Error <%=exception.getMessage() %> has occured.
</BODY>

</HTML>
```

Listing errorpage.jsp

Example using the JSP ErrorPage
Now it is time to utilse the above error page. For the our JSP page to be aware of the error page created above, use page directive as given in the following example. This page will throw an exception if invoked without data.

```
<% page errorPage="errorpage.jsp" %>
<%
    if(request.getQueryString()==null)
    {
     throw new Exception("Data required");
     }
%>
```

Listing TestErrorPage.jsp

Now call the above JSP page without a Query string to see the error page.

http://localhost:8080/examples/TestErrorPage.jsp

The JSP Include **Directive**

This directive lets you include files at the time the JSP page is translated into a servlet.

The directive looks like this:

<%@ include file="relative url" %>

The URL specified is normally interpreted relative to the JSP page that refers to it, but, as with relative URLs in general, you can tell the system to interpret the URL relative to the home directory of the Web server by starting the URL with a forward slash. The contents of the included file are parsed as regular JSP text, and thus can include static HTML, scripting elements, directives, and actions. For example, many sites include a small navigation bar on each page. Due to problems with HTML frames, this is usually implemented by way of a small table across the top of the page or down the left-hand side, with the HTML repeated for each page in the site. The include directive is a natural way of doing this, saving the developers from the maintenance nightmare of actually copying the HTML into each separate file. Here's some representative code:

Example

Now we are going to write one html file and two JSP files. This html file contains some coding which is common to both the JSP files. Instead of rewriting the contents we are going to include the entire file using 'include' directive in JSP.

File 1:

```
<HTML>
<BODY>
THIS CONTENT WILL BE DISPLAYED WHEREVER THE FILE GETS  INSERTED.
</BODY>
</HTML>
```

Listing Navigation.html

```
File 2:

<HTML>
<HEAD>
    <TITLE>JavaServer Pages (JSP) 1.0</TITLE>
</HEAD>

<BODY>
<H1> THIS IS JSP PAGE 1 </H1>
<%@ include file="/Navigation.html" %>
</BODY>

</HTML>
```

Listing JSPInclude1.jsp.

```
File 2:

<HTML>
<HEAD>
    <TITLE>JavaServer Pages (JSP) 1.0</TITLE>
</HEAD>

<BODY>
<H1> THIS IS JSP PAGE 1 </H1>
<%@ include file="/Navigation.html" %>
</BODY>
</HTML>
```

Listing JSPInclude1.jsp.

```
File3:

<HTML>
<HEAD>
    <TITLE>JavaServer Pages (JSP) 1.0</TITLE>
</HEAD>

<BODY>
<H1> THIS IS JSP PAGE 2 </H1>
<%@ include file="/Navigation.html" %>
</BODY>

</HTML>
```

Listing JSPInclude.jsp.

Note that since the "include directive" inserts the files at the time the page is translated, if the navigation bar changes, you need to re-translate all the JSP pages that refer to it. This is a good compromise in a situation like this, since the navigation bar probably changes infrequently, and you want the inclusion process to be as efficient as possible. If, however, the included files changed more often, you could use the jsp:include action(wee will see this soon) instead. This includes the file at the time the JSP page is requested.

Example Using Scripting Elements and Directives

Here is a simple example showing the use of JSP expressions, scriptlets, declarations, and directives.

```
<HTML>
<HEAD>
<TITLE>Using JavaServer Pages</TITLE>
</HEAD>
<BODY>

<H1> USING JSP PAGES </H1>
Some dynamic content created using various JSP mechanisms:
```

```
<UL>

<LI><B>Expression.</B><BR>

Your hostname: <%= request.getRemoteHost() %>.

<LI><B>Scriptlet. </B><BR>

<% out.println("Attached GET data: " +

        request.getQueryString()); %>

<LI><B>Declaration(plus expression).</B><BR>

<%! private int accessCount = 0; %>

Accesses to page since server reboot: <%= ++accessCount %>

<LI><B>Directive (plus expression).</B><BR>

<%@ page import = "java.util.*" %>


Current date: <%= new Date() %>

</UL>


</BODY>

</HTML>
```
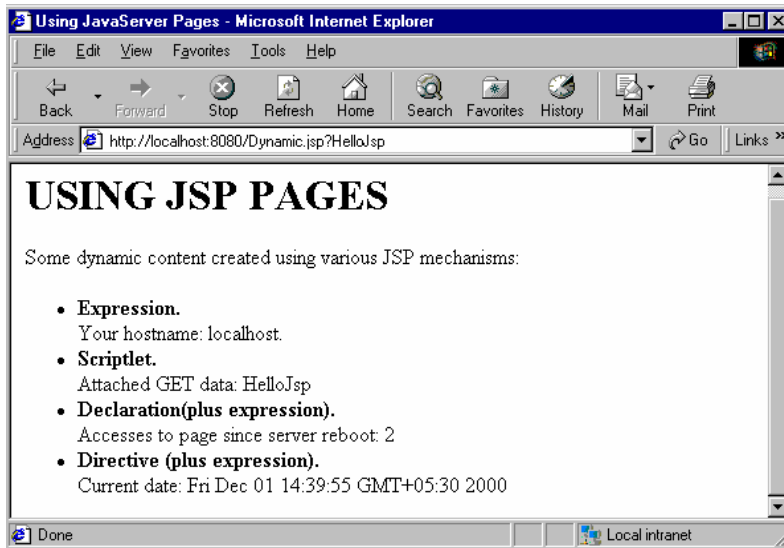
Listing Dynamic.jsp

Here's the typical result:

Figure 31.1 Java Server Pages

Syntax Summary

JSP Element	Syntax	Interpretation
JSP Expression	<%= expression %>	Expression is evaluated and placed in output.
JSP Scriptlet	<% code %>	Code is inserted in service method.
JSP Declaration	<%! code %>	Code is inserted in body of servlet class, outside of service method.
JSP page Directive	<%@ page att="val" %>	Directions to the servlet engine about general setup.
JSP include Directive	<%@ include file="url" %>	A file on the local system to be included when the JSP page is translated into a servlet.
JSP Comment	<%-- comment --%>	Comment; ignored when JSP page is translated into servlet.
The jsp:include Action	<jsp:include page="*relative URL*" flush="true"/>	Includes a file at the time the page is requested.
The jsp:useBean Action	<jsp:useBean att=val*/> or <jsp:useBean att=val*> ... </jsp:useBean>	Find or build a Java Bean.
The jsp:setProperty Action	<jsp:setProperty att=val*/>	Set bean properties, either explicitly or by designating that value comes from a request parameter.
The jsp:getProperty Action	<jsp:getProperty name="*propertyName*" value="*val*"/>	Retrieve and output bean properties.
The jsp:forward Action	<jsp:forward page="*relative URL*"/>	Forwards request to another page.
The jsp:plugin Action	<jsp:plugin attribute="*value*"*> ... </jsp:plugin>	Generates OBJECT or EMBED tags, as appropriate to the browser type, asking that an applet be run using the Java Plugin.

31.3 Predefined Variables

To simplify code in JSP expressions and scriptlets, you are supplied with eight automatically defined variables, sometimes called *implicit objects*. The available variables are request, response, out, session, application, config, pageContext, and page. Details for each are given below.

31.3.1 Request

This is the HttpServletRequest associated with the request, and lets you look at the request parameters (via getParameter), the request type (GET, POST, HEAD, etc.), and the incoming HTTP headers (cookies, Referrer, etc.). Strictly speaking, request is allowed to be a subclass of ServletRequest other than HttpServletRequest, if the protocol in the request is something other than HTTP. This is almost never done in practice.

31.3.2 Response

This is the HttpServletResponse associated with the response to the client. Note that, since the output stream (see out below) is buffered, it *is* legal to set HTTP status codes and response headers, even though this is not permitted in regular servlets once any output has been sent to the client.

31.3.3 Out

This is the PrintWriter used to send output to the client. However, in order to make the response object (see the previous section) useful, this is a buffered version of PrintWriter called JspWriter. Note that you can adjust the buffer size, or even turn buffering off, through use of the buffer attribute of the page directive. Also note that out is used almost exclusively in scriptlets, since JSP expressions automatically get placed in the output stream, and thus rarely need to refer to out explicitly.

Example using request and out objects.

This example just shows how a JSP file processes FORM data using request and out objects.

```
File 1 : A HTML file to collect and send  name and password data from the
user
<HTML>
<BODY >

<FORM TYPE=POST ACTION=result.jsp>
<FONT size=5 COLOR="red">

Type your name and password <br>
Name     <input TYPE=text NAME=name >  <BR>
Password <input TYPE=password NAME=password> <BR>
<INPUT TYPE=submit NAME=submit VALUE="Submit">

</FONT>
</FORM>
</BODY>
</HTML>
```

Listing text.html

```
File 2: A JSP file to process the data sent by text.html.

<HTML>
<BODY>
<%
  String name=request.getParameter("name");
  String password=request.getParameter("password");
  if(name.equals(""))
    out.println("Name is must..");
  if(password.equals(""))
    out.println("Password is must..");
 %>
</BODY>
</HTML>
```

Listing result.jsp

Session

This is the HttpSession object associated with the request. Recall that sessions are created automatically, so this variable is bound even if there was no incoming session reference. The HttpSession object is used to store objects in between client requests.

Example

This example illustrates the use of session object by tracking the number of visits the user makes. 'Refresh' many times and see how the value changes.

```
<HTML>
<BODY>

<%
//let 'session_count' be an object representing session
//   count
 Integer session_count =
       (Integer)session.getValue("COUNT");

// if session_count is not found then it is time to create  // one.

  if(session_count==null) {

    session_count = new Integer(1);
    session.putValue("COUNT",session_count);
  }

      // increment the session count otherwise.
      else {
        int new_value = session_count.intValue()+1;
        session_count = new Integer(new_value);
        session.putValue("COUNT",session_count);
      }

      out.println("This is "+ session_count +
                  " time you are visiting this site");
      %>

      </BODY>
      </HTML>
```

Listing Session.jsp

Application

This is the ServletContext as obtained via
getServletConfig().getContext().

The application object has application scope, which means that it is available to all JSPs until the JSP engine is shut down. The application object is most often used to retrieve environment information.

Config

This is the ServletConfig object for this page.

PageContext

JSP introduced a new class called PageContext to encapsulate use of server-specific features like higher performance JspWriters. The idea is that, if you access them through this class rather than directly, your code will still run on "regular" servlet/JSP engines.

Page

This is simply a synonym for this, and is not very useful in Java. It was created as a placeholder for the time when the scripting language could be something other than Java.

Actions

JSP *actions* provide an abstraction that can be used to easily encapsulate common tasks. They typically create or act on objects, normally JavaBeans. JSP *actions* use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin. Available actions include:

- jsp:include - Include a file at the time the page is requested.

- jsp:useBean - Find or instantiate a JavaBean.

- jsp:setProperty - Set the property of a JavaBean.

- jsp:getProperty - Insert the property of a JavaBean into the output.

- jsp:forward - Forward the requester to a new page.

- jsp:plugin - Generate browser-specific code that makes an OBJECT or EMBED tag for the Java plugin.

These actions are described in more detail below. Remember that, as with XML in general, the element and attribute names are case sensitive.

The Jsp:Include **Action**

This action lets you insert files into the page being generated. The syntax looks like this:

<jsp:include page="relative URL" flush="true" />

Unlike the include directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested. This pays a small penalty in efficiency, and precludes the included page from containing general JSP code (it cannot set HTTP headers, for example), but it gains significantly in flexibility. For example, here is a JSP page that inserts four different snippets into a "What's New?" Web page. Each time the headlines change, authors only need to update the four files, but can leave the main JSP page unchanged.

Example

```
<HTML>
<HEAD>
<TITLE>What's New</TITLE>
</HEAD>

<BODY>
<H1>What's New at JspNews.com</H1>
<P>

Here is a summary of our four most recent news stories:
<OL>
  <LI><jsp:include page="news/Item1.html" flush="true"/>
```

```
   <LI><jsp:include page="news/Item2.html" flush="true"/>
   <LI><jsp:include page="news/Item3.html" flush="true"/>
   <LI><jsp:include page="news/Item4.html" flush="true"/>
</OL>
</BODY>
</HTML>
```

Listing WhatsNew . jsp

Here's the typical result:

Figure 31.2 Actions for what's new at Jspnews

The jsp:useBean **Action**

This action lets you load in a JavaBean to be used in the JSP page. This is a a very useful capability because it lets you exploit the reusability of Java classes without sacrificing the convenience that JSP adds over Servlets alone. The simplest syntax for specifying that a bean should be used is:

```
<jsp:useBean id="name"
             class="package.class"
             scope="page|request|session|application"/>
```

This usually means "instantiate an object of the class specified by class, and bind it to a variable with the name specified by id." The scope attribute represents the life of the object.

The jsp:setProperty **Action**

You use jsp:setProperty to give values to properties of beans that have been referenced earlier. You can do this in two contexts. First, you can use jsp:setProperty after, but outside of, a jsp:useBean element, as below:

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName"
                 property="someProperty" ... />
```

In this case, the jsp:setProperty is executed regardless of whether a new bean was instantiated or an existing bean was found. A second context in which jsp:setProperty can appear is inside the body of a jsp:useBean element, as below:

```
<jsp:useBean id="myName" ... >
  ...
  <jsp:setProperty name="myName"
                   property="someProperty" ... />
</jsp:useBean>
```

Here, the jsp:setProperty is executed only if a new object was instantiated, not if an existing one was found.

There are four possible attributes of jsp:setProperty:

Attribute	Usage
Name	This required attribute designates the bean whose property will be set. The jsp:useBean element must appear before the jsp:setProperty element.
Property	This required attribute indicates the property you want to set. However, there is one special case: a value of "*" means that all request parameters

	whose names match bean property names will be passed to the appropriate setter methods.
Value	This optional attribute specifies the value for the property. String values are automatically converted to numbers, boolean, Boolean, byte, Byte, char, and Character via the standard valueOf method in the target or wrapper class. For example, a value of "true" for a boolean or Boolean property will be converted via Boolean.valueOf, and a value of "42" for an int or Integer property will be converted via Integer.valueOf. You can't use both value and param, but it is permissible to use neither. See the discussion of param below.
Param	This optional attribute designates the request parameter from which the property should be derived. If the current request has no such parameter, nothing is done: the system does *not* pass null to the setter method of the property. Thus, you can let the bean itself supply default values, overriding them only when the request parameters say to do so. For example, the following snippet says "set the numberOfItems property to whatever the value of the numItems request parameter is, if there is such a request parameter. Otherwise don't do anything." <jsp:setProperty name="orderBean" property="numberOfItems" param="numItems" /> If you omit both value and param, it is the same as if you supplied a param name that matches the property name. You can take this idea of automatically using the request property whose name matches the property one step further by supplying a property name of "*" and omitting both value and param. In this case, the server iterates through available properties and request parameters, matching up ones with identical names.

The jsp:getProperty **Action**

This element retrieves the value of a bean property, converts it to a string, and inserts it into the output. The two required attributes are name, the name of a bean previously referenced via jsp:useBean, and property, the property whose value should be inserted.

Now we are going to see one sample application which use jsp:useBean, jsp:setAttribute and jsp:getAttribute actions. This example does nothing but manipulation of a string via JavaBean.

Example using Beans inside a JSP page

File 1: Bean program

```
package hall;

public class SimpleBean {
private String message = "No message specified";

//Bean's getter method.
public String getMessage() {
  return(message);
}

// Bean's setter method.
public void setMessage(String message) {
  this.message = message;
}
}
```

Listing SimpleBean.java

File2: BeanJSP.jsp

```
<HTML>
<BODY>

<jsp:useBean id="test" class="hall.SimpleBean" />
<jsp:setProperty name="test"
            property="message"
                   value="Hello WWW" />
<H1> Message:
<I>  <jsp:getProperty name="test" property="message" /></I>
</H1>
```
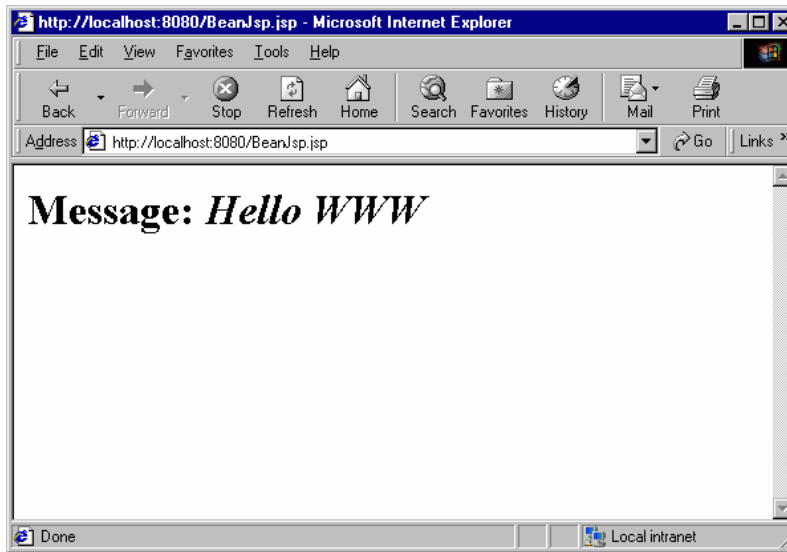
```
</BODY>
</HTML>
```

Lising BeanJsp.jsp

To work with the above example,

- The bean .class should reside in the javawebserver2.0\classes\hall directory

- If your .jsp file is inside javawebserver2.0\public_html directory then type the following
  in the URL of the browser:
  http://localhost:8080/BeanJsp.jsp

Here's the typical output:

31.4 Redirecting to an exeternal page <Jsp:request>

JSP defines a tag, <jsp:request>, that you can use to redirect to an external page in one of two ways, as specified by the FORWARD attribute or the INCLUDE attribute. With the FORWARD attribute, you can redirect to any valid URL. This effectively halts processing of the current page at the point where the redirect occurs, although all processing up to that point will still take place. This is exactly analogous to a typical redirect using CGI, SSJS, ASP, or JavaScript. With the INCLUDE attribute, you can not only redirect to another page but

also come back to the calling page upon completion of processing in the called page. For instance, you could actually call out to another JSP page that generates some HTML dynamically and have that page generate its HTML; upon returning, that HTML would be inserted into the calling page at the point where your <jsp:request> tag occurs. In fact, the called page has no idea that it's being called from another JSP page. It simply sees an HTTP request and responds by returning HTML text.

Keep in mind that you can use the INCLUDE method of redirection to access static HTML pages, JSP pages, servlets, SSJS pages, ASP pages - just about any resource that responds to HTTP requests and generates a response that you want to include in your page. But note that if the resource you access returns a complete HTML page, including <HTML> and <BODY> tags, you may not get the result you intended.

The jsp:forward **Action**

This action lets you forward the request to another page. It has a single attribute, page, which should consist of a relative URL. This could be a static value, or could be computed at request time, as in the two examples below.

<jsp:forward page="/utils/errorReporter.jsp" />
<jsp:forward page="<%= someJavaExpression %>" />

The jsp:plugin **Action**

This action lets you insert the browser-specific OBJECT or EMBED element needed to specify that the browser run an applet using the Java plugin.

Comments and Character Quoting Conventions

There are a small number of special constructs you can use in various cases to insert comments or characters that would otherwise be treated specially. Here's a summary:

Syntax	Purpose
<%-- comment --%>	A JSP comment. Ignored by JSP-to-scriptlet translator. Any embedded JSP scripting elements, directives, or actions are ignored.

<!-- comment -->	An HTML comment. Passed through to resultant HTML. Any embedded JSP scripting elements, directives, or actions are executed normally.
<\%	Used in template text (static HTML) where you really want "<%".
%\>	Used in scripting elements where you really want "%>".
\'	A single quote in an attribute that uses single quotes. Remember, however, that you can use either single or double quotes, and the other type of quote will then be a regular character.
\"	A double quote in an attribute that uses double quotes. Remember, however, that you can use either single or double quotes, and the other type of quote will then be a regular character.
%\>	%> in an attribute.
<\%	<% in an attribute.

31.5 Short summary

- Translation time Errors and Request Time Errors are the two types of JSP error .

- The available variables in JSP is request, response, out, session, application, config, pageContext, and page. Details for each are given below.

31.6 Brain Storm

1. How to handle JSP error?

2. What are the various predefined variables in JSP?

ಸ End ಜ

`

Lecture 32

Discussion

Manonmaniam Sundaranar University
Centre for Information Technology and Engineering
Tirunelveli

Syllabus for MS(IT&EC) / MIT

2.4 Advanced Java Programming with Database Application

Lecture 1	DBMS Introduction – Summary of DBMS function – Codd's Rules
Lecture 2	SQL – Using SQL as DDL,DML and Data Query Language – Functions
Lecture 3	JDBC Architecture - Remote Database Access.
Lecture 4	JDBC Introduction – Connecting to an ODBC Data Source – JDBC Connection – JDBC Implementation – Resultset Processing.
Lecture 5	JDBC Prepared statement – Callable Statement – other JDBC classes – Moving Cursor in crollable Result Sets – Making updates to updateable Result Sets – Updating a Result Set programmatically.
Lecture 6	Introduction to software component - Software component model - Java Bean - Importance of Java Bean – Bean Development kit
Lecture 7	Building simple bean - Event Handling
Lecture 8	Bean persistence – Serialization and Deserialization
Lecture 9	Introspection
Lecture 10	Properties - simple, Boolean, indexed, bound properties
Lecture 11	Properties - constrained – customizations
Lecture 12	Discussion
Lecture 13	EJB overview – client/server architecture – component transaction monitors – middle ware architecture – application server – examples – application server – transactional and tier view – middle ware and 3 tier view
Lecture 14	Need of EJB – What is EJB – EJB architecture – EJB features- deployment – Roles and responsibilities
Lecture 15 &16	Creating a simple EJB - Implementation – looking into the working
Lecture 17	Introduction to distributed application – Introduction to RMI- RMI Architecture – Boot strapping and the RMI registry – working of RMI – Advantages of RMI

Lecture 18	Steps involved in creating client applications
Lecture 19	Dynamic class loading – Introduction- codebase in applets, RMI – command line examples – example of dynamic class loading
Lecture 20	Trouble shooting tips – problem while running the RMI server and RMI client – object activation
Lecture 21	Making an object activatable (the remote interface – implementation class – policy file – creating "setup" class – compile and run the code).
Lecture 22	Discussion
Lecture 23	Introduction – CGI-Servelet overview – Basic Servlet Structure – Examples
Lecture 24	Handling form data –Introduction - request headers
Lecture 25	Response headers – overviews – common response headers – examples
Lecture 26	Cookies overview – The Servlet coolie API-creating, reading $ specifying cookie attributes – cookie utilities
Lecture 27	Session tracking – introduction – session tracking API- Servlet communications – calling servlets from servlets
Lecture 28	Working with URLs - reading directly from a URL – reading from and writing to a URL connection
Lecture 29	JSP Basics – advantages of JSP – JSP request model
Lecture 30	JSP Architecture – getting with JSP – components of JSP – JSP scripting elements – JSP scriplets – JSP declarations – JSP directives
Lecture 31	Handling JSP errors - creating JSP error page – examples using scripting elements & directives – predefined variables – comments and character quoting conventions.
Lecture 32	Discussion

ℬ Best of Luck ℭ