

Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Learning Objectives

After successfully completing the module, students will be able to do the following:

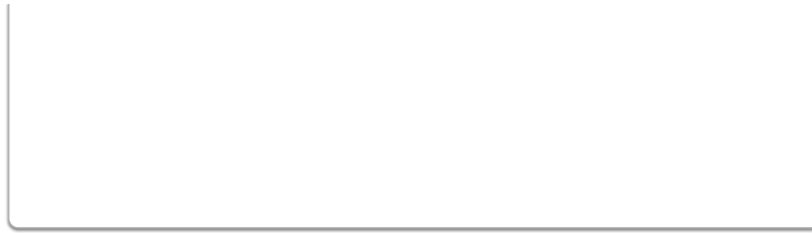
1. Describe relational database concepts.
2. Use SQL to create and drop tables and also to retrieve and modify data.
3. Use the JDBC API to access databases and process *ResultSets*.
4. Use *PreparedStatement*s to execute pre-compiled SQL statements.
5. Use *CallableStatements* to execute stored SQL procedures.
6. Use the *DatabaseMetaData* and *ResultSetMetaData* interfaces.
7. Apply the concepts of TCP, IP, and Internet address.
8. Create servers using server sockets and clients using client sockets.
9. Implement Java networking programs using stream sockets.
10. Develop servers for multiple clients.
11. Send and receive objects on a network.

Module 6 Study Guide and Deliverables

- Topics:
- Lecture 6 Java Database Connectivity
 - Lecture 7 Java Networking Connectivity
- Readings:
- Deitel & Deitel, Chapter 24
 - Networking: Deitel & Deitel, Chapter 28 (online; accessible from the textbook's web site)
 - Module 6 online content
- Assessments: Interim Assessment: none
- Assignments: Assignment 6 due Wednesday, February 23 at 6:00 AM ET
- Live Classrooms:
- Wednesday, February 16, from 8:00 PM to 9:00 PM ET
 - Thursday, February 17, from 9:00 PM to 10:00 PM ET

Module Welcome and Introduction

met_cs622_18_su1_ebraude_mod6 video cannot be displayed here



■ Lecture 6 – Java Database Connectivity

Relational Database

A database is a repository or storehouse of data. The data in a database contain information about items of interest. A database management system (DBMS) is the collection of software that stores and manages data in a database. Application programs are generally built on top of the DBMS for users to access the database. Most of the present-day DBMSs are relational database management systems or RDBMS. RDBMSs are based on the "relational" data model.

A relational data model is based on the concept of data being stored in relations or tables. In an RDBMS, a "relation" and a "table" are taken to be synonymous. Conceptually, a table is a two-dimensional construct consisting of rows and columns. Each table has a name so that it can be identified. Each column has a name and stores one piece of information. Each row one set of values for all columns in the table. As an example, here is a table for recording information about people.

Person Relational Table

first_name	last_name	birth_date
Bob	Smith	3/27/1974
Jane	Glass	5/19/1999
Vishnu	Santhana	12/13/1950

There is a first_name, last_name, and birth_date column in the table, to store a person's first name, last name, and birth date, respectively. There are three rows in the table. The first row is for Bob Smith, born 3/27/1974. The second is for Jane Glass, born 5/19/1999. The third is for Vishnu Santhana, born 12/13/1950. You can see by looking at this example, each row stores information about one person. Rows of a table are also referred to as tuples or records. Generally, each record in a table contains information about one real or abstract object or thing.

Each table represents a real or abstract concept. For example, the Person table represents the concept of a person. Each row represents one individual person. You can also think of a table as a collection of real or abstract objects. For example, the Person table contains a collection of people. An RDBMS often contains multiple tables, each with its own name.

Every column has a datatype which specifies the type and maximum size of the values in the column. The permissible values are defined by the datatype. Three common datatypes are VARCHAR, DECIMAL, and DATE. VARCHAR allows for a series of characters up to a maximum length. DECIMAL allows for numbers, and whether decimal points are allowed, and how many digits are allowed, is customizable. DATE allows for dates. There are other datatypes, and every database also has its own specific datatypes, but these three are very common.

The relational model also supports relationships between rows. When the same data value is duplicated in two different rows of the same or different tables, those rows are related. Typically, rows in different tables are related, such as between a Person and Address. However, in some cases rows in the same table are related. For example, an Employee can be related to another employee through a "manages" relationship since a manager is also an employee.

In modern times, we are used to working with spreadsheet applications. Each table is akin to a worksheet in a spreadsheet application.

Integrity Constraints

Integrity constraints provide conditions that must at all times be satisfied by the values in the tables. Relational databases support 5 kinds of constraints on tables – primary key, foreign key, not null, unique, and check. These low-level constraints are used to ensure basic data integrity. Violating these constraints would make the data invalid.

Primary key constraint:

A primary key is a column or set of columns that uniquely identifies every row. Primary key values are used as the basis for creating a reference to a table. You can think of a primary key as the "address" of a table row. A primary key constraint enforces the fact that the column(s) are unique and not null. For example, A `person_id` column in a `Person` table can be assigned a primary key constraint.

Foreign key constraint:

A foreign key constraint enforces the fact that a reference is valid. Each value specified in a foreign key column must exist in the table being referenced. For example, If an `Employee` table has a foreign key to an `Address` table via an `address_id` column, any value in the `address_id` column in `Employee` must exist in `Address` (i.e. the reference must be valid).

Not null constraint:

A not null constraint requires that every row in the table must have a value for that column. For example, often the `last_name` column in a `Person` table is not null so that a last name is required for every person.

Unique constraint:

A unique constraint requires that every value in the table for the column or set of columns must be unique (that is, each value must appear only once). Contrary to popular belief, a unique constraint does not require the column also have a not null constraint. For example, If a `person` table already has a `person_id` primary key, but also has a `social_security_number` field, the `social_security_number` field can be given a unique constraint to ensure every person's number is unique. This would catch bad data entry, for example.

Check constraint:

A check constraint supports for a Boolean expression that can involve any of the table's columns, to check for cross-column integrity. The other four constraints apply a narrow condition to a column or set of columns. A check constraint applies any Boolean expression to any or all columns in a table.

For example, imagine that an `Employee` table has an `hourly_rate` column for hourly employees, and a `yearly_salary` column for salaried employees. Any particular employee will have a value in only one of these, since they will either be an hourly employee, or a salaried employee. A check constraint can be used to ensure that one of the fields must have a value, but both fields do not simultaneously have a value. None of the other constraints make this possible.

Integrity constraints are continually enforced by the database management system. The database aggressively rejects actions that would violate these constraints.

Relational/Object-Oriented Isomorphism

The relational model and the object-oriented model have several isomorphic concepts. It's useful for programmers to understand how tables and relationships in a relational database are isomorphic with classes and references.

Table/Class Isomorphism

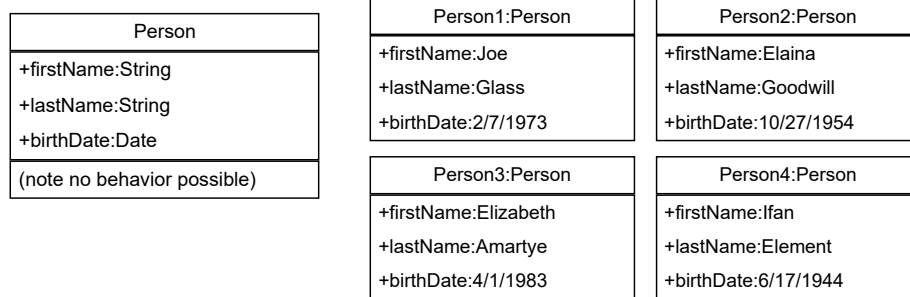
A relational table is isomorphic with a Java class that has no behavior, that is, has no methods. There is no concept of table methods in a relational database table; tables store data. Table columns are isomorphic with public member variables of a class. There is no concept of protected, package, or private data in a relational table, so effectively all columns are "public". Table rows are isomorphic with objects of a class. One row is one object.

Let's look at the example below of a Person table and how it translates to the object-oriented model.

Person Table

first_name	last_name	birth_date
Joe	Glass	2/7/1973
Elaina	Goodwill	10/27/1954
Elizabeth	Amartye	4/1/1983
Ifan	Element	6/17/1944

Isomorphic With



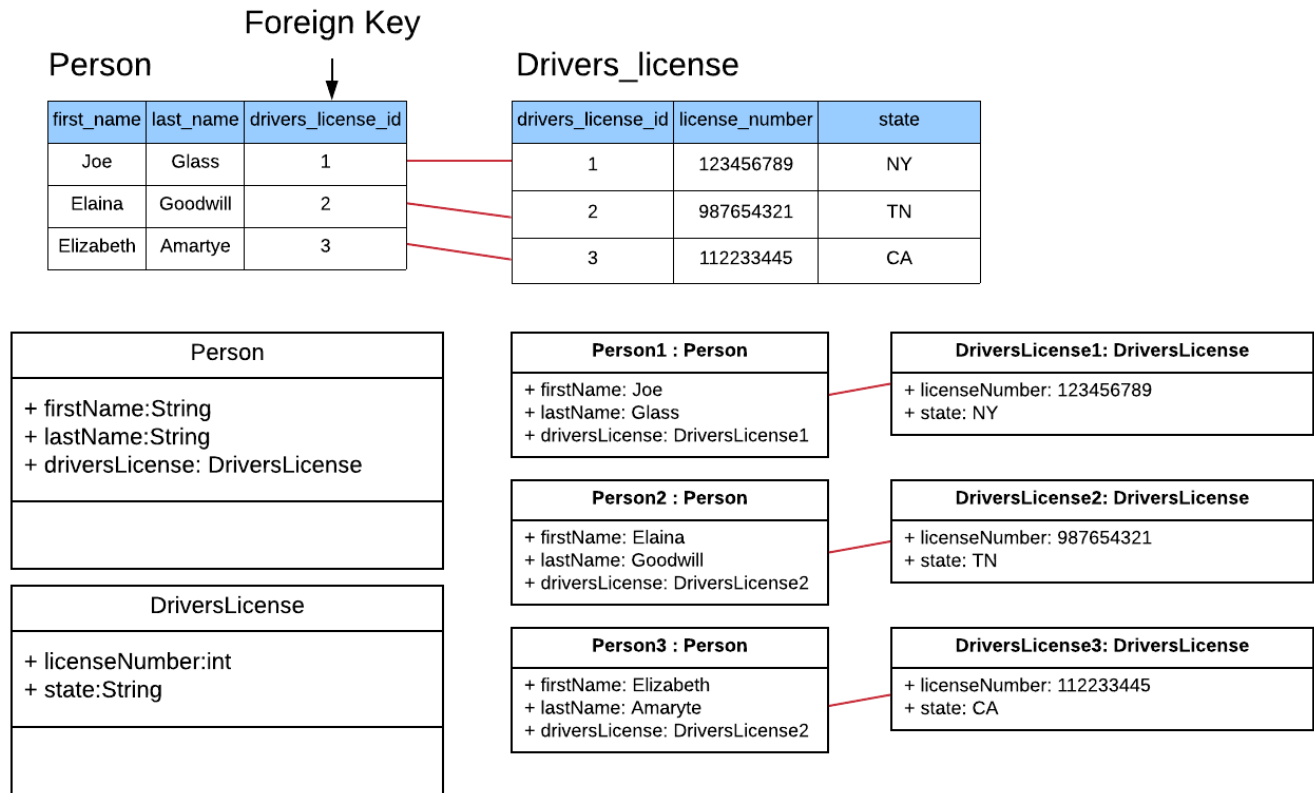
The Person table has three columns, first_name, last_name, and birth_date. An isomorphic Person class would have three public member variables – firstName, lastName, and birthDate – with no methods in the class. The Person table has four rows, and these would be four objects in the object-oriented model, one for each row, as the diagram illustrates.

Relationship/Reference Isomorphism

Relationships in the relational model are created by duplicating data values. For example, if a value in one row the value “5”, and a value in another row has the value “5”, we can consider these rows as related. Relationships are enforced through foreign-key constraints, essentially to ensure that the relationship is valid.

Relationships in the object-oriented model are both created and enforced through references. One object can reference another object.

The example below illustrates that relationships and references are isomorphic.



In the example, the Person table is related to the Drivers_license table through a foreign key on drivers_license_id. This is equivalent to a Person class having a reference to a DriversLicense class in the object-oriented model. Notice that in the relational model, the values that create the relationship are accessible. We can see that driver's license id 1 exists in both tables, for example. We can access and change these values as well. In the object-oriented model, the underlying address that creates the reference is not accessible. We can change which object is being referenced, but cannot see or modify the internal mechanics of the reference.

A Publishing Database

In what follows, we create and use a SQLite database named Module6Lecture.db (more about the SQLite relational database appears later in this module). It is a publishing database. The database contains information on researchers, journals and publications.

- Researchers write up the results of their research in the form of papers that appear in published form in journals. Information stored about a researcher includes his or her ID (unique), first name and last name.
- A journal is represented by its ID (unique), journal title and the publisher's name.
- A given publisher may publish multiple (different) journals. A given journal is published by one and only one publisher. A given researcher can publish (different) papers in different journals. A given researcher never publishes the same paper in more than one journal. A given journal publishes papers by different researchers. A given journal never publishes a given paper more than once.

The database design may include the following relations (tables):

- researcher
 - Attributes: rid, fname, lname
 - Primary key: rid
- journal
 - Attributes: jid, title, publisher
 - Primary key: jid
- publication
 - Attributes: rid, jid, quantity
 - Primary key: rid and jid taken together

- Foreign key: rid (from Table Researcher), jid (from Table Journal)

The quantity field in the publication table stores the number of papers published by a given researcher in a given journal. It is conceivable that with time, as a researcher publishes new papers in a journal in which he/she had previously published, the quantity field will be incremented. The section on SQL (later in this module) explains how to create these tables from scratch and how to load them with data.

SQLite Overview

We use the SQLite database to illustrate database connectivity in Java throughout this lecture. Before proceeding with the rest of this lecture, you'll want to follow the steps in the [Getting Started with SQLite](#) tutorial. That tutorial will get you up and running with SQLite and its associated SQL client, DB Browser for SQLite.

SQLite is the most used, embedded (serverless) relational database in the world. It is open source and free to use. Unlike server-based databases like Oracle and SQL Server, SQLite runs entirely in the application that uses it, and stores all of its durable objects in a single disk file. SQLite can be used across all major platforms, which means the database file can be freely copied and used across devices with different architectures. SQLite is ideal for applications that would traditionally use files to store data, giving them access to the power of a relational database without the expense and overhead of installing and maintaining a server-based database.

Once you learn to access and use any one modern relational database, you can use the others without much additional effort. All modern relational databases utilize the Structured Query Language (SQL) for data access and manipulation. SQL is highly standardized across databases. Although there are some differences, the significant aspects are the same across databases. In addition, Java supports a standardized API, JDBC, for accessing any database. Connectivity from Java does not differ much between databases. Thus SQLite is an excellent first database for Java developers, because the intricacies of relational databases and connectivity can be learned without the overhead of database installation, yet SQLite is used in serious applications worldwide.

DB Browser for SQLite Overview

It is a best practice to manage your database's structure with a SQL client. Typically, we use a SQL client to first add the tables, indexes, and triggers (if needed), as well as any initial data. Then when our application executes, it will add, modify, and remove data as needed with JDBC, but not modify the structure of the tables and indexes. By separating structure manipulation from data manipulation, we can carefully apply good database design principles, and avoid embedding table structure in our application.

A popular SQL client for SQLite is DB Browser for SQLite. The [Getting Started with SQLite](#) tutorial shows you how to install it and create a table with it.

In this lecture, we use DB Browser for SQLite on publishingdb to illustrate several frequently used features of SQL. The following types of SQL statements will be discussed:

- Create and destroy tables: create; drop
- Insert, delete, update tuples: insert; delete; update
- Queries on single tables: select; select distinct; select with aggregate functions; select with the following clauses:
 - where
 - order by
 - group by
 - having
- Queries on multiple tables

SQL

SQL (pronounced "sequel" or "S Q L"), or *structured query language*, is a general language for creating, modifying, and accessing tables in an RDBMS. Different RDBMSs use essentially the same SQL language (with minor variations); in this sense SQL is a universal database language.

Create and Drop

The SQL statement to create a table specifies the table name, attributes (columns or fields), and types of the attributes. The following statement creates a table named "researcher":

```
create table researcher (
    rid int not null,
    fname varchar(20) not null,
    lname varchar(30) not null,
    primary key (rid)
);
```

In the above example, the researcher table has three attributes: "rid" (researcher ID), "fname" (first name), and "lname" (last name), with rid specified as the primary key. The statement also stipulates that none of the attributes can have a null value. The rid field is an integer, while fname and lname are variable-length strings that can be up to 20 and 30 characters long, respectively.

The above create statement (and other SQL statements) can be executed within a SQL client such as DB Browser for SQLite, or from within a Java program (to be discussed later in this module). It is a best practice to manage your database's structure with a SQL client, and run the data commands in your application. This means commands like CREATE, ALTER, and DROP are typically executed in a SQL client, because they are one-time commands, and commands like SELECT, INSERT, UPDATE, and DELETE are typically executed within the application, because they are executed over and over again as the data changes.

The tables journal and publication are created as follows:

```
create table journal (
    jid int not null,
    title varchar(30) not null,
    publisher varchar(20) not null,
    primary key (jid)
);
create table publication (
    rid int not null,
    jid int not null,
    quantity int not null,
    primary key (rid, jid),
    foreign key (rid) references researcher(rid),
    foreign key (jid) references journal(jid)
);
```

The publication table is specified as having a (composite) primary key consisting of the fields rid and jid, which are the same as the identically-named fields in tables researcher and journal, respectively. We say that the rid field in publication references the rid field in researcher; similarly, the jid field in publication references the jid field in journal. The two fields rid and jid, therefore, are foreign keys in publication.

When a table is no longer needed, it can be destroyed (dropped permanently) using the drop statement. For instance, the following statement drops the table named researcher:

```
drop table researcher;
```

If the table to be dropped is referenced by other tables, then those other tables must first be dropped. For instance, having created the researcher, journal and publication tables by using the above SQL statements, if we wish to drop all the three tables, publication must be dropped before researcher and journal can be dropped.

Insert

Once a table is created, we can insert data (one or more records) into it by using the insert statement. The following example inserts seven records into the researcher table:

```
insert into researcher
values
(10, 'David', 'Goldberg'),
(70, 'James', 'Smith'),
(30, 'David', 'Kaiser'),
(40, 'John', 'Doe'),
(50, 'Greg', 'Rawlins'),
```

```
(20,'Henry','Kimura'),
(60,'John','Smith');
```

Similarly, the journal table is loaded using the SQL statement:

```
insert into journal
values
(501,'Photonics','IEEE'),
(701,'Nature','Macmillan'),
(301,'Power Sources','Elsevier'),
(401,'Physical Review','APS'),
(101,'Quantum Electronics','IEEE'),
(201,'Computer','IEEE'),
(801,'Energy','Elsevier'),
(601,'Information Sciences','Elsevier'),
(901,'Nature Photonics','Macmillan');
```

And four records are stored in table publication by using the following insert statement:

```
insert into publication
values
(10, 501, 1),
(20, 701, 5),
(60, 101, 1),
(20, 401, 2);
```

Queries

We will now use the data in these three tables to illustrate the various SQL queries. We are using the DB Browser for SQLite SQL client to demonstrate the results.

The Select Statement

The (ordinary) select statement retrieves the specified fields (attributes) from all the rows of the specified table. For example, the query

```
select title, publisher
from journal;
```

returns the following information:

	title	publisher
1	Photonics	IEEE
2	Nature	Macmillan
3	Power Sources	Elsevier
4	Physical Review	APS
5	Quantum Electronics	IEEE
6	Computer	IEEE
7	Energy	Elsevier
8	Information Sciences	Elsevier
9	Nature Photonics	Macmillan

These results are reflected in the following screenshot:

SQL 1 

```

1  select title, publisher
2  from journal;
3
4

```

	title	publisher
1	Photonics	IEEE
2	Nature	Macmillan
3	Power Sources	Elsevier
4	Physical Review	APS
5	Quantum Electronics	IEEE
6	Computer	IEEE
7	Energy	Elsevier
8	Information Sciences	Elsevier
9	Nature Photonics	Macmillan

Using * in the Select Statement

Instead of one or more attributes, a star (*) can be specified to retrieve all the attributes. The query

```
select * from journal;
```

produces the following output:

```

      jid      title      publisher
1 501  Photonics      IEEE
2 701   Nature      Macmillan
3 301  Power Sources  Elsevier
4 401  Physical Review APS
5 101  Quantum Electronics IEEE
6 201   Computer      IEEE
7 801   Energy      Elsevier
8 601  Information Sciences Elsevier
9 901  Nature Photonics Macmillan

```

These results are reflected in the following screenshot:

SQL 1 

```

1  select * from journal
2
3
4

```

	jid	title	publisher
1	501	Photonics	IEEE
2	701	Nature	Macmillan
3	301	Power Sources	Elsevier
4	401	Physical Review	APS
5	101	Quantum Electronics	IEEE
6	201	Computer	IEEE
7	801	Energy	Elsevier
8	601	Information Sciences	Elsevier
9	901	Nature Photonics	Macmillan

Select distinct


The "select distinct" feature selects distinct (non-repeating) values, as in the following example:

```
select distinct publisher
from journal;
```

Output:

```
publisher
1 IEEE
2 Macmillan
3 Elsevier
4 APS
```

These results are reflected in the following screenshot:

SQL 1 

```
1 select distinct publisher
2 from journal;
3
4
```

	publisher
1	IEEE
2	Macmillan
3	Elsevier
4	APS

Using the "where" Clause in the Select Statement


The "where" clause specifies a condition that the selected tuples must satisfy. For instance, the following query retrieves only those records that correspond to at least two published papers:

```
select * from publication
where quantity >= 2;
```

Output:

```
rid  jid  quantity
1 20   701   5
2 20   401   2
```

These results are reflected in the following screenshot:

SQL 1 

```
1 select * from publication
2 where quantity >= 2;
3
4
```

	rid	jid	quantity
1	20	701	5
2	20	401	2

The Operator "like" and the Percent (%) Character


The operator "like" is used in the where clause to match patterns. The percent (%) character matches zero or more characters. Therefore the following query retrieves all researchers whose first name starts with a 'J':

```
select * from researcher
where fname like 'J%';
```

Output:

```
rid  fname  lname
1 70   James   Smith
2 40   John    Doe
3 60   John    Smith
```

These results are reflected in the following screenshot:

SQL 1 

```

1 select * from researcher
2 where fname like 'J%';
3
4

```

	rid	fname	lname
1	70	James	Smith
2	40	John	Doe
3	60	John	Smith

The underscore (_)

The underscore (_) matches exactly one character. The following query outputs all researchers whose last name has 'a' as the second character from the left:

```


select * from researcher
where lname like '_a%';

```

Output:

	rid	fname	lname
1	30	David	Kaiser
2	50	Greg	Rawlins

These results are reflected in the following screenshot:

SQL 1 

```

1 select * from researcher
2 where lname like '_a%';
3
4

```

	rid	fname	lname
1	30	David	Kaiser
2	50	Greg	Rawlins

The following statement produces all researchers whose last name has 'r' in the second place from the right:

```


select * from researcher
where lname like '%r_';

```

Output:

	rid	fname	lname
1	10	David	Goldberg
2	20	Henry	Kimura

These results are reflected in the following screenshot:

SQL 1 

```

1 select * from researcher
2 where lname like '%r_';
3
4

```

	rid	fname	lname
1	10	David	Goldberg
2	20	Henry	Kimura

Using the "order by" Clause in the Select Statement


The output of a query can be produced on sorted order (on one or more fields) by using the "order by" clause. The default order is ascending. The following query produces a list of researchers sorted on ascending order of their lastname:

```
select * from researcher
order by lname;
```

Output:

	rid	fname	lname
1	40	John	Doe
2	10	David	Goldberg
3	30	David	Kaiser
4	20	Henry	Kimura
5	50	Greg	Rawlins
6	70	James	Smith
7	60	John	Smith

These results are reflected in the following screenshot:

SQL 1 

```

1 select * from researcher
2 order by lname;
3
4
```

	rid	fname	lname
1	40	John	Doe
2	10	David	Goldberg
3	30	David	Kaiser
4	20	Henry	Kimura
5	50	Greg	Rawlins
6	70	James	Smith
7	60	John	Smith


The same result is obtained with the following statement:

```
select * from researcher
order by lname asc;
```

Output:

	rid	fname	lname
1	40	John	Doe
2	10	David	Goldberg
3	30	David	Kaiser
4	20	Henry	Kimura
5	50	Greg	Rawlins
6	70	James	Smith
7	60	John	Smith

These results are reflected in the following screenshot:

SQL 1 

```

1 select * from researcher
2 order by lname asc;
3
4
```

	rid	fname	lname
1	40	John	Doe
2	10	David	Goldberg
3	30	David	Kaiser
4	20	Henry	Kimura
5	50	Greg	Rawlins
6	70	James	Smith
7	60	John	Smith

Combining the "where" and "order by" Clauses

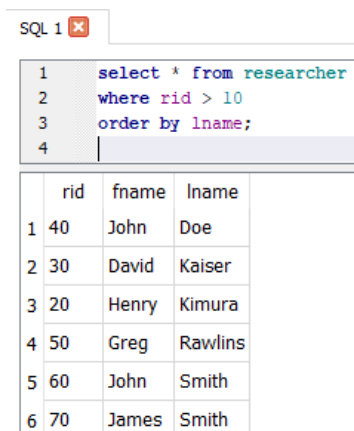
The "where" and "order by" clauses can be combined:

```
select * from researcher
where rid > 10
order by lname;
```

The sorting is done on only the records with rid greater than 10, producing the following output:

	rid	fname	lname
1	40	John	Doe
2	30	David	Kaiser
3	20	Henry	Kimura
4	50	Greg	Rawlins
5	60	John	Smith
6	70	James	Smith

These results are reflected in the following screenshot:



The screenshot shows a SQL query editor with a query window titled "SQL 1" containing the following SQL code:

```
1 select * from researcher
2 where rid > 10
3 order by lname;
4
```

Below the query editor, the results are displayed in a table with 6 rows and 3 columns: rid, fname, and lname.

	rid	fname	lname
1	40	John	Doe
2	30	David	Kaiser
3	20	Henry	Kimura
4	50	Greg	Rawlins
5	60	John	Smith
6	70	James	Smith

Specifying "desc" in the "order by" Clause

The descending order in sorting can be obtained by specifying "desc" in order by:

```
select * from researcher
order by lname desc;
```

Output:

	rid	fname	lname
1	70	James	Smith
2	60	John	Smith
3	50	Greg	Rawlins
4	20	Henry	Kimura
5	30	David	Kaiser
6	10	David	Goldberg
7	40	John	Doe

These results are reflected in the following screenshot:

SQL 1 

```

1 select * from researcher
2 order by lname desc;
3

```

	rid	fname	lname
1	70	James	Smith
2	60	John	Smith
3	50	Greg	Rawlins
4	20	Henry	Kimura
5	30	David	Kaiser
6	10	David	Goldberg
7	40	John	Doe

The following statement sorts the records on ascending order of lname, and within lname, on ascending order of fname:

```

select * from researcher
order by lname, fname;

```

Output:

```

rid  fname  lname
1 40   John   Doe
2 10   David  Goldberg
3 30   David  Kaiser
4 20   Henry  Kimura
5 50   Greg   Rawlins
6 70   James  Smith
7 60   John   Smith

```

These results are reflected in the following screenshot:

SQL 1 

```

1 select * from researcher
2 order by lname, fname;
3
4

```

	rid	fname	lname
1	40	John	Doe
2	10	David	Goldberg
3	30	David	Kaiser
4	20	Henry	Kimura
5	50	Greg	Rawlins
6	70	James	Smith
7	60	John	Smith

The following statement produces sorting on descending order of lname, and within lname, on ascending order of fname (James followed by John):

```

select * from researcher
order by lname desc, fname asc;

```

Output:

```

rid  fname  lname
1 70   James  Smith
2 60   John   Smith
3 50   Greg   Rawlins
4 20   Henry  Kimura
5 30   David  Kaiser
6 10   David  Goldberg
7 40   John   Doe

```

These results are reflected in the following screenshot:

SQL 1 ✕

```

1  select * from researcher
2  order by lname desc, fname asc;
3
4

```

	rid	fname	lname
1	70	James	Smith
2	60	John	Smith
3	50	Greg	Rawlins
4	20	Henry	Kimura
5	30	David	Kaiser
6	10	David	Goldberg
7	40	John	Doe

The following operators can be used in the where clause:

Operators in the where Clause

Operator	Meaning
not	Logical negation
and	Logical conjunction
or	Logical disjunction
=	Equal to
!= (or <>)	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Select with Aggregate Functions

The following five aggregate functions can be used in a select command:

- count
- min
- max
- sum
- avg

Each of these functions produces a single value from a group of values. For example, the statement

```

select count(*), min(quantity), max(quantity), sum(quantity), avg(quantity)
from publication;

```

produces the following output:

```

count(*) min(quantity) max(quantity) sum(quantity) avg(quantity)
1 4          1          5          9          2.25

```

These results are reflected in the following screenshot:

SQL 1 

```

1 select count(*), min(quantity), max(quantity), sum(quantity), avg(quantity)
2 from publication;
3
4

```

	count(*)	min(quantity)	max(quantity)	sum(quantity)	avg(quantity)
1	4	1	5	9	2.25

where the serial numbers 1 through 5 in the column headings indicate the five items selected for retrieval. The results are self-explanatory (4 is the total count of records in the table, the minimum value of quantity from the four rows is 1, the maximum is 5, and so on).

We could make the output more readable by specifying suitable names to be used as column headings, as follows:

```

select count(*) as totalcount, min(quantity) as minimum, max(quantity) as maximum,
sum(quantity) as grouptotal, avg(quantity) as average from publication;

```

Output:

```

totalcount minimum maximum grouptotal average
1 4          1          5          9          2.25

```

These results are reflected in the following screenshot:

SQL 1 

```

1 select count(*) as totalcount, min(quantity) as minimum, max(quantity) as maximum,
2 sum(quantity) as grouptotal, avg(quantity) as average from publication;
3

```

	totalcount	minimum	maximum	grouptotal	average
1	4	1	5	9	2.25

Group by

The "group by" clause forms groups of rows (records) with the same column (field) value. The query

```

select publisher
from journal
group by publisher;

```

produces the following output:

```

publisher
1 APS
2 Elsevier
3 IEEE
4 Macmillan

```

These results are reflected in the following screenshot:

SQL 1 

```

1 select publisher
2 from journal
3 group by publisher;
4

```

	publisher
1	APS
2	Elsevier
3	IEEE
4	Macmillan

This is so because the above query produces four groups, one for each of the four different publishers, and then produces a single row of result from each group. The items specified in the select statement must each be single-valued per group.

As another example, consider

```

select publisher, count(publisher) as count
from journal

```




```
group by publisher
order by count;
```

Output:

	publisher	count
1	APS	1
2	Macmillan	2
3	Elsevier	3
4	IEEE	3

These results are reflected in the following screenshot:

SQL 1 

```

1 select publisher, count(publisher) as count
2 from journal
3 group by publisher
4 order by count;
5
```

	publisher	count
1	APS	1
2	Macmillan	2
3	Elsevier	3
4	IEEE	3

Having Clause


The having clause can be used with the group by clause to filter out one or more groups. Thus it acts like a condition that a group must satisfy for inclusion in the final result (recall that the where clause conditionally filters out rows). For example, the following example eliminates from the final result groups for which the publisher publishes fewer than two journals:

```
select publisher, count(publisher) as count
from journal
group by publisher
having count(publisher) > 1;
```

Output:

	publisher	count
1	Elsevier	3
2	IEEE	3
3	Macmillan	2

These results are reflected in the following screenshot:

SQL 1 

```

1 select publisher, count(publisher) as count
2 from journal
3 group by publisher
4 having count(publisher) > 1;
5
```

	publisher	count
1	Elsevier	3
2	IEEE	3
3	Macmillan	2

Multi-table Queries: Join

SQL allows us to combine information from more than one table. This is accomplished through the use of columns (attributes) that are common to multiple tables. For instance, all the researchers in the publication table (indicted by their rid field) must be present in the researcher table. (Note, however, that not all researchers in the researcher table are present in the publication table; only those who published a paper in a journal are.) Thus we can join the researcher table and the publication table on the rid attribute. Similarly, the jid attribute joins the two tables journal and publication.


To find the names (first and last) of all researchers who have journal publications (and the journal titles in which they published), we use the following SQL query:

```
select fname, lname, title
from researcher, journal, publication
where journal.jid = publication.jid and researcher.rid = publication.rid;
```

The output is given by:

	fname	lname	title
1	David	Goldberg	Photonics
2	Henry	Kimura	Nature
3	Henry	Kimura	Physical Review
4	John	Smith	Quantum Electronics

These results are reflected in the following screenshot:

SQL 1 

```

1 select fname, lname, title
2 from researcher, journal, publication
3 where journal.jid = publication.jid and researcher.rid = publication.rid;
4

```

	fname	lname	title
1	David	Goldberg	Photonics
2	Henry	Kimura	Physical Review
3	Henry	Kimura	Nature
4	John	Smith	Quantum Electronics


Similarly, the following query retrieves all researchers (first and last names) who have published in an IEEE journal:

```
select fname, lname, title
from publication
join researcher ON researcher.rid = publication.rid
join journal ON journal.jid = publication.jid
where journal.publisher = 'IEEE';
```

Output:

	fname	lname	title
1	David	Goldberg	Photonics
2	John	Smith	Quantum Electronics

These results are reflected in the following screenshot:

SQL 1 

```

1 select fname, lname, title
2 from publication
3 join researcher ON researcher.rid = publication.rid
4 join journal ON journal.jid = publication.jid
5 where journal.publisher = 'IEEE';
6

```

	fname	lname	title
1	David	Goldberg	Photonics
2	John	Smith	Quantum Electronics

Update and Delete

The Update Statement

The update statement allows one or more tuples to be updated, usually conditionally. For example, the following statement causes all instances of 'Macmillan' to be changed to 'Macmillan UK'.

```
update journal
set publisher = 'Macmillan UK'
where publisher = 'Macmillan';
```

A "select * from journal" command would now produce the following:

	jid	title	publisher
1	501	Photonics	IEEE
2	701	Nature	Macmillan UK
3	301	Power Sources	Elsevier
4	401	Physical Review	APS
5	101	Quantum Electronics	IEEE
6	201	Computer	IEEE
7	801	Energy	Elsevier
8	601	Information Sciences	Elsevier
9	901	Nature Photonics	Macmillan UK

These results are reflected in the following screenshot:

The screenshot shows a SQL query editor with a toolbar at the top containing 'SQL 1' and a red 'X' icon. Below the toolbar, the query 'select * from journal' is entered on line 1. Line 2 is empty. Below the query editor, the results are displayed in a table with 3 columns: jid, title, and publisher. The table contains 9 rows of data, matching the output shown in the previous block.

	jid	title	publisher
1	501	Photonics	IEEE
2	701	Nature	Macmillan UK
3	301	Power Sources	Elsevier
4	401	Physical Review	APS
5	101	Quantum Electronics	IEEE
6	201	Computer	IEEE
7	801	Energy	Elsevier
8	601	Information Sciences	Elsevier
9	901	Nature Photonics	Macmillan UK

The Delete Statement

The delete statement deletes, usually conditionally, one or more tuples. For example, the following statement

```
delete from journal
where title = 'Energy';
```

deletes the row corresponding to the journal Energy. The result can be seen by issuing a "select * from journal" query:

	jid	title	publisher
1	501	Photonics	IEEE
2	701	Nature	Macmillan UK
3	301	Power Sources	Elsevier
4	401	Physical Review	APS
5	101	Quantum Electronics	IEEE
6	201	Computer	IEEE
7	601	Information Sciences	Elsevier
8	901	Nature Photonics	Macmillan UK

These results are reflected in the following screenshot:

SQL 1 

```
1 select * from journal
2
3
```

	jid	title	publisher
1	501	Photonics	IEEE
2	701	Nature	Macmillan UK
3	301	Power Sources	Elsevier
4	401	Physical Review	APS
5	101	Quantum Electronics	IEEE
6	201	Computer	IEEE
7	601	Information Sciences	Elsevier
8	901	Nature Photonics	Macmillan UK

SQL - Summary

Of the SQL statements discussed in this module, create and drop statements are examples of SQL data definition language (DDL), while select, insert, delete, and update are examples of data manipulation language (DML).

Accessing a Database with JDBC

JDBC, or Java database connectivity, is the Java API for developing database applications in Java. JDBC provides programmers a mechanism to access relational databases through Java programs. By using JDBC, Java programs can execute SQL statements, and retrieve data from the database into Java program variables and store data from program variables into the database.

The JDBC API is database agnostic. Programmers don't need to learn a new API for every database; they learn this one. Each database vendor provides a JDBC driver that applies the API calls to their specific database. Once you learn how to use JDBC with one database, you can use it for any other (keeping in mind that there are some minor differences in the SQL language for different databases).

As mentioned previously in this lecture, the modern best practice is to modify the structure of your tables (typically using CREATE, ALTER, and DROP commands) in a SQL client, and to use the app to keep the data updated (typically using SELECT, INSERT, UPDATE, and DELETE commands). The structure is set once, then the data regularly changes over time based upon the application execution.

For illustrating JDBC usage, we have created a SQLite database named "Book.db". We created a table called "authors" where each author has an ID (unique key) and a firstname (fname) and a lastname (lname), using the SQL below.

```
create table authors (
    id int not null,
    fname varchar(20) not null,
    lname varchar(30) not null,
    primary key (id)
);
```

We have inserted six records (rows) corresponding to as many authors, using the SQL below.

```
insert into authors
values
(1, 'Charles', 'Dickens'),
(2, 'Bibhutibhushan', 'Bandyopadhyay'),
(3, 'John', 'Muir'),
(4, 'Mark', 'Twain'),
(5, 'John', 'Fitzgerald'),
(6, 'Gerald', 'Durrell');
```

Typical JDBC Sequence

Code that accesses a database using JDBC follows this sequence.

1. A database connection is established.
2. A Statement (or child of Statement) object is created with SQL embedded in the statement.
3. The Statement object, and thus the SQL embedded in the Statement object, is executed.
4. Results from the statement execution are handled as needed.
5. The statement object is closed.
6. Steps 2-5 are repeated as needed for other statements.
7. The database connection is closed.

This series of steps is how Java applications throughout the world use their database.

Step 1 establishes a connection. JDBC provides a Connection class for this purpose. Steps 2-5 use the JDBC provided Statement class, or child thereof, to execute and retrieve the results from SQL. Then ultimately the database connection is closed.

JDBC Connection

JDBC requires a highly structured connection string to create a connection to a database. The connection information includes at a minimum the type of database (technically, the type of driver such as Oracle, SQLite, SQL Server, etc.) and where the database can be accessed.

For example, to connect to our Book.db database using SQLite, this is the connection string:

```
jdbc:sqlite:C:/SQLite/GettingStarted.db
```

The prefix "jdbc" is always used because we are always using JDBC. The next segment, "sqlite", identifies which database driver to load. Each JDBC driver has a specific string to identify itself, and for SQLite it's "sqlite". The next segment identifies the database location. For SQLite, this is the path to the database file. For networked databases such as Oracle, SQL Server, and Postgres, this would not be a file path, but the IP address and port to connect to. For our use of SQLite, these three options are all that's needed.

For other databases such as Oracle, SQL Server, and Postgres, it's common to have a few database-specific option, which are represented as key/value pairs separated by semicolons. One common option is "database=<database_name>" to identify which database to connect to (assuming the instance has multiple databases). Other common options include "user=<username>" and "password=<password>", to specify account credentials. We don't need to use these for our SQLite purposes, however, but it's good to be aware of them so you understand what they are for if you see them in code examples.

To obtain the connection, the *DriverManager* class is used to return a Connection object, like so:

```
String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
try (Connection conn = DriverManager.getConnection(connectionString)) {
}
```

The connection is obtained in a try/with block so that it is automatically closed once the code block completes. Establishing a connection is the first step, and after this SQL can be executed against the database.

JDBC Statement

JDBC provides a Statement class with subclasses to execute SQL. These are JDBC's container for SQL statements.

We discuss three types of Java statements that are used for accessing a database:

- **Statement:** used with no parameters
- **PreparedStatement:** used with only input parameters
- **CallableStatement:** used with both input and output parameters

The program below illustrates how, by using the JDBC "Statement," we can issue an SQL statement from within a Java program. The SQL command we use in this example is a simple select statement to retrieve all the records from the authors table.

```

package database;
import java.sql.*;

public class BasicStatement {
    public static void main(String[] args) throws SQLException {
        String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
        try (Connection connection = DriverManager.getConnection(connectionString)) {
            System.out.println("Database now connected.");
            try (Statement statement = connection.createStatement()) {
                ResultSet results =
                    statement.executeQuery("SELECT id, fname, lname FROM Authors");
                System.out.println("Here are the authors:");
                while (results.next()) {
                    System.out.printf("%d\t%-10s\t%-10s\n",
                        results.getInt(1), results.getString(2), results.getString(3));
                }
            }
        }
    }
}

```

The following statement creates a Connection object to connect to the Book database.

```
Connection connection = DriverManager.getConnection(connectionString)
```

Once the Connection object is created, we create a statement object for executing a SQL statement, by using the connection object, as follows:

```
Statement statement = connection.createStatement()
```

A SQL query statement can be executed with the executeQuery(String) statement, where the String argument is an SQL DML query, as in the following example:

```

ResultSet results =
    statement.executeQuery("SELECT id, fname, lname FROM Authors");

```

In the above example, the executeQuery() method returns as the result of the query an object that implements interface ResultSet. The ResultSet object contains the result of the SQL query. The ResultSet object can be thought of as representing a table that has multiple rows in it, each row containing information retrieved about each author in the authors table. ResultSet's methods can now be used to retrieve the results.

We use the next() method of ResultSet to move to the next row (the initial row position is null), in the below.

```
while (results.next()) {
```

Each iteration in the loop moves to the next row, and if there are no more rows, the next() method returns false to terminate the loop.

The ResultSet interface has "get" methods to obtain the values in each row. There is a different "get" method for each datatype, including getString() and getInt(). We use these methods in the code below.

```

System.out.printf("%d\t%-10s\t%-10s\n",
    results.getInt(1), results.getString(2), results.getString(3));

```

The getInt(1), getString(2), and getString(3) methods retrieve the attribute values for ID, fname and lname, respectively, from the authors table. The column ordinal is specified with each method call. Note that the first column ordinal is 1 and is not 0, unlike arrays and lists.

The results of calling this program are listed below, the full set of authors.

```

Database now connected.
Here are the authors:
1   Charles      Dickens
2   Bibhutibhushan Bandyopadhyay
3   John         Muir
4   Mark         Twain
5   John         Fitzgerald
6   Gerald       Durrell

```

These results are reflected in the following screenshot:

```

1 package database;
2 import java.sql.*;
3
4 public class BasicStatement {
5     public static void main(String[] args) throws SQLException {
6         String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
7         try (Connection connection = DriverManager.getConnection(connectionString)) {
8             System.out.println("Database now connected.");
9             try (Statement statement = connection.createStatement()) {
10                 ResultSet results =
11                     statement.executeQuery("SELECT id,fname,lname FROM Authors");
12                 System.out.println("Here are the authors:");
13                 while (results.next()) {
14                     System.out.printf("%d\t%-10s\t%-10s\n",
15                         results.getInt(1), results.getString(2), results.getString(3));
16                 }
17             }
18         }
19     }
20 }
21

```

Problems | Javadoc | Declaration | Console

<terminated> BasicStatement [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (Apr 15, 2020, 12:2)

Database now connected.
Here are the authors:

1	Charles	Dickens
2	Bibhutibhushan	Bandyopadhyay
3	John	Muir
4	Mark	Twain
5	John	Fitzgerald
6	Gerald	Durrell

Note that an alternative way to retrieve the same three values is to use the column names themselves: `getInt(id)`, `getString(fname)`, and `getString(lname)`, as shown below.

```

System.out.printf("%d\t%-10s\t%-10s\n",
    results.getInt("id"), results.getString("fname"), results.getString("lname"));

```

PreparedStatement

The prior Statement interface example used a static SQL statement that had no parameters. The PreparedStatement interface is more powerful in that it can be used with SQL statements with or without parameters. The following program illustrates the use of a PreparedStatement which executes a similar select statement, but retrieves only the rows that satisfy the conditions in a where clause condition.

```

package database;
import java.sql.*;

public class PreparedStatementExample {
    public static void main(String[] args) throws SQLException {
        String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
        try (Connection connection = DriverManager.getConnection(connectionString)) {
            System.out.println("Database now connected.");
            String sql =
                "SELECT id,fname,lname " +
                "FROM authors " +
                "WHERE id > ? AND fname = ?";
            try (PreparedStatement statement = connection.prepareStatement(sql)) {
                statement.setInt(1, 2);
                statement.setString(2, "John");
                ResultSet results =
                    statement.executeQuery();
                System.out.println("Here are the limited authors:");
                while (results.next()) {
                    System.out.printf("%d\t%-10s\t%-10s\n",
                        results.getInt(1), results.getString(2), results.getString(3));
                }
            }
        }
    }
}

```

```
    }
}
```

The where clause is written with two input parameters, indicated by the question mark (?) symbols. In the above example, a PreparedStatement object is created with the Connection interface's prepareStatement(String) method:

```
String sql =
    "SELECT id, fname, lname " +
    "FROM authors " +
    "WHERE id > ? AND fname = ?";
try (PreparedStatement statement = connection.prepareStatement(sql)) {
```

PreparedStatement is a sub-interface of Statement, and provides setter methods for setting the parameters in objects of PreparedStatement. The two setter methods:

```
statement.setInt(1, 2);
statement.setString(2, "John");
```

supply the values 2 (an int) for id and "John" (a String) for fname in the select statement. The syntax for setting the parameters is

```
setType(int parameterIndex, Type value);
```

where parameterIndex is the index of the parameter in the statement (it starts from 1). The values of the parameters can be set statically or dynamically.

Upon execution, the above program prints out the rows corresponding to John Muir and John Fitzgerald, as shown below.

```
Database now connected.
Here are the limited authors:
3   John      Muir
5   John      Fitzgerald
```

These results are reflected in the following screenshot:

```
1 package database;
2 import java.sql.*;
3
4 public class PreparedStatementExample {
5     public static void main(String[] args) throws SQLException {
6         String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
7         try (Connection connection = DriverManager.getConnection(connectionString)) {
8             System.out.println("Database now connected.");
9             String sql =
10                 "SELECT id, fname, lname " +
11                 "FROM authors " +
12                 "WHERE id > ? AND fname = ?";
13             try (PreparedStatement statement = connection.prepareStatement(sql)) {
14                 statement.setInt(1, 2);
15                 statement.setString(2, "John");
16                 ResultSet results =
17                     statement.executeQuery();
18                 System.out.println("Here are the limited authors:");
19                 while (results.next()) {
20                     System.out.printf("%d\t%-10s\t%-10s\n",
21                                     results.getInt(1), results.getString(2), results.getString(3));
22                 }
23             }
24         }
25     }
26 }
27
```

<terminated> PreparedStatementExample [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (Apr 15, 2020, 12:37:12 PM)

```
Database now connected.
Here are the limited authors:
3   John      Muir
5   John      Fitzgerald
```

Obtaining Metadata

JDBC allows us to access metadata, or data on the database itself. Information on the database management system (DBMS) is often useful. We discuss two types of metadata:

- Database metadata

- Resultset metadata

Database Metadata

Database information (metadata) can easily be obtained by using the DatabaseMetaData interface. See the following example.

```
package database;
import java.sql.*;

public class BasicMetadata {
    public static void main(String[] args) throws SQLException {
        String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
        try (Connection connection = DriverManager.getConnection(connectionString)) {
            System.out.println("Database now connected.");
            DatabaseMetaData dbmd = connection.getMetaData();

            System.out.println("URL: " + dbmd.getURL());
            System.out.println("User name: " + dbmd.getUserName());
            System.out.println("Product name: " + dbmd.getDatabaseProductName());
            System.out.println("Product version: " + dbmd.getDatabaseProductVersion());
            System.out.println("Driver name: " + dbmd.getDriverName());
            System.out.println("Driver version: " + dbmd.getDriverVersion());
        }
    }
}
```

In order to obtain an instance of DatabaseMetaData for a database, we use the getMetaData method on a Connection object:

```
DatabaseMetaData dbmd = connection.getMetaData();
```

The DatabaseMetaData methods getURL(), getUserName(), etc. shown in the above program are self-explanatory.

The above program produces the following output:

```
Database now connected.
URL: jdbc:sqlite:C:/SQLite/Book.db
User name: null
Product name: SQLite
Product version: 3.30.1
Driver name: SQLite JDBC
Driver version: 3.30.1
```

These results are reflected in the following screenshot:

```

1 package database;
2 import java.sql.*;
3
4 public class BasicMetadata {
5     public static void main(String[] args) throws SQLException {
6         String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
7         try (Connection connection = DriverManager.getConnection(connectionString)) {
8             System.out.println("Database now connected.");
9             DatabaseMetaData dbmd = connection.getMetaData();
10
11             System.out.println("URL: " + dbmd.getURL());
12             System.out.println("User name: " + dbmd.getUserName());
13             System.out.println("Product name: " + dbmd.getDatabaseProductName());
14             System.out.println("Product version: " + dbmd.getDatabaseProductVersion());
15             System.out.println("Driver name: " + dbmd.getDriverName());
16             System.out.println("Driver version: " + dbmd.getDriverVersion());
17         }
18     }
19 }
20

```

Problems @ Javadoc Declaration Console

```

<terminated> BasicMetadata [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (Apr 15, 2020, 12:5
Database now connected.
URL: jdbc:sqlite:C:/SQLite/Book.db
User name: null
Product name: SQLite
Product version: 3.30.1
Driver name: SQLite JDBC
Driver version: 3.30.1

```

ResultSet Metadata

The `ResultSetMetaData` interface provides information on the `ResultSet`. The following example shows how to run a simple query and obtain information from the resulting `ResultSet` object.

```

package database;
import java.sql.*;

public class ResultSetMetadata {
    public static void main(String[] args) throws SQLException {
        String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
        try (Connection connection = DriverManager.getConnection(connectionString)) {
            System.out.println("Database now connected.");
            try (Statement statement = connection.createStatement()) {
                ResultSet results =
                    statement.executeQuery("SELECT id, fname, lname FROM Authors");
                ResultSetMetaData rsmd = results.getMetaData();
                System.out.println("Here are the column names for the query.");
                for (int i = 1; i <= rsmd.getColumnCount(); i++) {
                    System.out.printf("%-20s\n", rsmd.getColumnName(i));
                }
            }
        }
    }
}

```

In the above program, the

```
ResultSetMetaData rsmd = results.getMetaData();
```

statement creates a `ResultSetMetaData` object called `rsmd`. The number of columns in the result set is obtained with the `getColumnCount()` method of `ResultSetMetaData`. The `getColumnName(i)` methods returns the column name at column `i` (with the indexing starting at 1).

```

Database now connected.
Here are the column names for the query.
id
fname
lname

```

These results are reflected in the following screenshot:

```

1 package database;
2 import java.sql.*;
3
4 public class ResultSetMetadata {
5     public static void main(String[] args) throws SQLException {
6         String connectionString = "jdbc:sqlite:C:/SQLite/Book.db";
7         try (Connection connection = DriverManager.getConnection(connectionString)) {
8             System.out.println("Database now connected.");
9             try (Statement statement = connection.createStatement()) {
10                 ResultSet results =
11                     statement.executeQuery("SELECT id,fname,lname FROM Authors");
12                 ResultSetMetadata rsmd = results.getMetaData();
13                 System.out.println("Here are the column names for the query.");
14                 for (int i = 1; i <= rsmd.getColumnCount(); i++) {
15                     System.out.printf("%-20s\n", rsmd.getColumnName(i));
16                 }
17             }
18         }
19     }
20 }

```

Problems @ Javadoc Declaration Console

<terminated> ResultSetMetadata [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (Apr 15, 2020, 1:10:10 PM)

Database now connected.
 Here are the column names for the query.
 id
 fname
 lname

Lecture 7 – Java Networking Connectivity

Networking Overview

Computers connected in a network can communicate with one another. Java supports two modes of communication:

- stream-based (that uses *transfer control protocol*, or TCP, for data transmission), and
- packet-based (that uses *user datagram protocol*, or UDP)

Stream-based communication is used more commonly in Java programming and is discussed in this module.

Client-server Mode of Computing

Simply stated, in client-server computing, two computers, each running one program, communicate. We say that the two computers communicate, or equivalently, two programs communicate. One of the two communicating computers is called the server, while the other is referred to as the client. The idea is that the client computer (program) requests the server computer (program) for some service. Typically, one server services requests from multiple clients, often at the same time.

Server Socket and Client Socket

Two Java programs (running on two different computers in a network) communicate through two *sockets*. Sockets can be thought of as the end-points of the connection between two computers. Sockets are used to send and receive data. Java provides the `ServerSocket` class for creating a server socket and the `Socket` class for creating a client socket.

To create a server, a Java program needs to create a server socket and attach it to a *port*. For instance, the following statement creates a server socket named "serversocket" and attaches it to port number 9000:

```
ServerSocket serversocket = new ServerSocket(9000);
```

After creating the server socket, the server waits for a connection using the following statement:

```
Socket socket = serversocket.accept();
```

The server waits until a client connects to the server socket. The client uses the following statement to request a connection to the server:

```
Socket socket = new Socket(servername, port);
```

where servername is the IP (Internet Protocol) address or Internet host name of the server.

The host name "localhost" and the IP address 127.0.0.1 are special in that a program can use them to refer to the (local) machine on which a client is running.

Below we show an example of client-server communication, by writing two programs – a server program (running on the server machine) and a client program (running on the client machine). In our example, we run both the server program and the client program on the same machine ("localhost"), and we assume that the port that the server and client use is numbered 9000.

This example illustrates a very simple case of the client accepting a number from the user and then using the server to obtain the square of the number, which is reported back to the user. The server and the client exchange data through input/output streams on the socket. Typically, a server runs indefinitely and serves multiple clients. For simplicity, the present example shows a single execution of the server program and a single client.

The server program (class SingleServer1):

```
import java.io.*;
import java.net.*;
import java.util.*;

public class SingleServer1
{
    public static void main(String[] args)
    {
        try
        {
            ServerSocket serversocket = new ServerSocket(9000);
            System.out.println("Server says: Server started ... waiting ...");
            Socket socket = serversocket.accept();

            DataInputStream inputfromclient = new DataInputStream(socket.getInputStream());
            DataOutputStream outputtoclient = new DataOutputStream(socket.getOutputStream());

            int j = inputfromclient.readInt();
            System.out.printf("Server says: client sent %d\n", j);

            outputtoclient.writeInt(j * j);
            System.out.printf("Server says: client got back %d\n", j * j);

        }

        catch (IOException excep)
        {
            excep.printStackTrace();
        }
    }
}
```

The client program (class SingleClient1):

```
import java.io.*;
import java.net.*;
import java.util.*;
public class SingleClient1
{
    public static void main(String[] args)
    {
        try
        {
```

```

        System.out.println("Client says: client started");
        Socket socket = new Socket("localhost", 9000);

        DataInputStream fromserver = new DataInputStream(socket.getInputStream());
        DataOutputStream toserver = new DataOutputStream(socket.getOutputStream());

        Scanner input = new Scanner(System.in);
        System.out.print("Enter an integer that you want squared: ");
        int i = input.nextInt();

        System.out.printf("Client says: client passed %d to server\n", i);
        toserver.writeInt(i);
        int j = fromserver.readInt();
        System.out.printf("Client says: client received %d as the square from server\n", j);
    }

    catch (IOException excep)
    {
        excep.printStackTrace();
    }
}
}

```

In this example, the server starts and listens for a client. (The server must be running when a client tries to connect to the server.) A client connects to the server, and sends the number 5 to the server. The server computes the result (the square of 5, or 25) and sends the result back to the client. Note that if wrapping to `DataInputStream` and `DataOutputStream` is not done, then we have to stay confined to the raw byte-only read/write of `InputStream` and `OutputStream`.

When two command windows are opened on the same computer, and the server and client programs executed simultaneously (the server having been started first, followed by the client), we have the following sample session:

Output of the program – the server program

```

Administrator: Command Prompt
C:\Users\uday\UdayJava\network>javac SingleServer1.java
C:\Users\uday\UdayJava\network>java SingleServer1
Server says: Server started ... waiting ...
Server says: client sent 5
Server says: client got back 25
C:\Users\uday\UdayJava\network>

```

Output of the program – the client program

```

Administrator: Command Prompt
C:\Users\uday\UdayJava\network>javac SingleClient1.java
C:\Users\uday\UdayJava\network>java SingleClient1
Client says: client started
Enter an integer that you want squared: 5
Client says: client passed 5 to server
Client says: client received 25 as the square from server
C:\Users\uday\UdayJava\network>

```

InetAddress

The `InetAddress` class can be used to find the client's host name and IP address. The `getHostName()` and `getHostAddress()` methods retrieve the name and the address, respectively.

A server can find out the host name and IP address of the client by first obtaining an `InetAddress` object from a socket that connects to a client:

```

InetAddress inetadd = socket.getInetAddress();

```

and then invoking the `getHostName()` and `getHostAddress()` methods on the object `inetadd`.

The following example illustrates creating an `InetAddress` object by using `InetAddress`'s static method `getByName()`. The first argument on the command line is assumed to be a hostname.

```
import java.net.*;

public class TestInetAddress
{
    public static void main(String[] args)
    {
        try
        {
            InetAddress inetad = InetAddress.getByName(args[0]);
            System.out.println("IP add = " + inetad.getHostAddress()
                               + " Host name = " + inetad.getHostName());
        }

        catch (UnknownHostException excep)
        {
            System.out.println("Unknown IP address or host " + args[0]);
        }
    }
}
```

Output of the program – `InetAddress`

```
Administrator: Command Prompt
C:\Users\uday\UdayJava\network>javac TestInetAddress.java
C:\Users\uday\UdayJava\network>java TestInetAddress localhost
IP add = 127.0.0.1 Host name = localhost
C:\Users\uday\UdayJava\network>java TestInetAddress www.bu.edu
IP add = 128.197.26.35 Host name = www.bu.edu
C:\Users\uday\UdayJava\network>
```

Single Server and Multiple Clients

A single server typically serves more than one client at the same time. Each client is handled by one thread in a multi-thread program. In the following example, we assume that the server runs continuously, only to be stopped by a control-c from the user.

The server socket can have multiple connections. Each pass through the `while(true)` loop in the following server program creates a new connection. For each connection, the server starts a new thread to handle communication between the server and a new client. The client program is the same as the one used earlier in this module.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class MultiThreadServer
{
    public static void main(String[] args)
    {
        try
        {
            ServerSocket serversocket = new ServerSocket(9000);
            System.out.println("Server says: Server started ... waiting ...");

            while (true)
            {

```

```

        Socket socket = serversocket.accept();

        InetAddress inetad = socket.getInetAddress();
        System.out.println("Client IP address: " + inetad.getHostAddress() +
            " host name: " + inetad.getHostName());

        new Thread(new ProcessSingleClient(socket)).start();
    }
}

catch (IOException excep)
{
    excep.printStackTrace();
}
}

class ProcessSingleClient implements Runnable
{
    private Socket socket;

    public ProcessSingleClient(Socket s)
    {
        socket = s;
    }

    public void run()
    {
        try
        {
            System.out.println("Thread Id = " + Thread.currentThread().getId() + " start");
            DataInputStream inputfromclient = new DataInputStream(socket.getInputStream());
            DataOutputStream outputtoclient = new DataOutputStream(socket.getOutputStream());

            int j = inputfromclient.readInt();
            System.out.printf("Server says: client sent %d\n", j);

            outputtoclient.writeInt(j * j);
            System.out.printf("Server says: client got back %d\n", j * j);
            System.out.println("Thread Id = " + Thread.currentThread().getId() + " end");
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}

```

A sample run of the server program with two simultaneous clients and then with a single client is shown below. Note that the computation for the inputs of 5000 and 17 are interleaved (Thread 12, corresponding to the input of 99, starts after both Thread 10 and Thread 11 have finished).

Output of the program – server program with two simultaneous clients and then with a single client

```

Administrator: Command Prompt
C:\Users\uday\UdayJava\network>javac MultiThreadServer.java
C:\Users\uday\UdayJava\network>java MultiThreadServer
Server says: Server started ... waiting ...
Client IP address: 127.0.0.1 host name: MET-CHAKRABORTY.ad.bu.edu
Thread Id = 10 start
Client IP address: 127.0.0.1 host name: MET-CHAKRABORTY.ad.bu.edu
Thread Id = 11 start
Server says: client sent 5000
Server says: client got back 25000000
Thread Id = 10 end
Server says: client sent 17
Server says: client got back 289
Thread Id = 11 end
Client IP address: 127.0.0.1 host name: MET-CHAKRABORTY.ad.bu.edu
Thread Id = 12 start
Server says: client sent 99
Server says: client got back 9801
Thread Id = 12 end
C:\Users\uday\UdayJava\network>

Administrator: Command Prompt
C:\Users\uday\UdayJava\network>javac SingleClient1.java
C:\Users\uday\UdayJava\network>java SingleClient1
Client says: client started
Enter an integer that you want squared: 17
Client says: client passed 17 to server
Client says: client received 289 as the square from server
C:\Users\uday\UdayJava\network>

Administrator: Command Prompt
C:\Users\uday\UdayJava\network>javac SingleClient2.java
C:\Users\uday\UdayJava\network>java SingleClient2
Client2 says: client2 started
Enter an integer that you want squared: 5000
Client2 says: client2 passed 5000 to server
Client2 says: client2 received 25000000 as the square from server

C:\Users\uday\UdayJava\network>java SingleClient2
Client2 says: client2 started
Enter an integer that you want squared: 99
Client2 says: client2 passed 99 to server
Client2 says: client2 received 9801 as the square from server
C:\Users\uday\UdayJava\network>

```

Transferring Objects through Sockets

For sending and receiving objects, the client and the server have to use `ObjectInputStream` and `ObjectOutputStream`. The following example shows a client sending to the server employee data; the server updates the employee's salary by computing a 10% bonus, and then sends the modified employee object back to the client.

Since we are using `ObjectInputStream` and `ObjectOutputStream`, the `Employee` class has to be *Serializable*. In this example, we repeat the `Employee` class definition in the server program because the server needs access to the salary field of employees.

```

import java.io.*;
import java.net.*;
import java.util.*;

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

```



```

try
{
    ServerSocket serversocket = new ServerSocket(9000);
    System.out.println("Server says: Server started ... waiting ...");
    Socket socket = serversocket.accept();

    ObjectInputStream inputfromclient = new ObjectInputStream(socket.getInputStream());
    ObjectOutputStream outputtoclient = new ObjectOutputStream(socket.getOutputStream());

    Object obj = inputfromclient.readObject();
    if (obj instanceof Employee)
    {
        Employee em = (Employee) obj;
        System.out.printf("Server says: client sent this object: %s\n", em);
        em.addbonus();
        outputtoclient.writeObject(em);
    }
    else
    {
        System.out.printf("Server says: client sent a non-employee - returned to client\n");
        outputtoclient.writeObject(obj);
    }
}

catch (ClassNotFoundException excep)
{
    excep.printStackTrace();
}

catch (IOException excep)
{
    excep.printStackTrace();
}
}

class Employee implements Serializable
{
    private int id;
    private double salary;

    public Employee(int empid, double sal)
    {
        id = empid;
        salary = sal;
    }

    public void addbonus()
    {
        salary *= 1.1; // 10% bonus added
    }

    public String toString()
    {
        return "id = " + id + " and salary = " + salary;
    }
}

```

The client program is given below:

```

import java.io.*;
import java.net.*;
import java.util.*;

```

```

public class ClientTransferObject
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Client says: client started");
            Socket socket = new Socket("localhost", 9000);

            ObjectOutputStream toserver = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream fromserver = new ObjectInputStream(socket.getInputStream());

            Scanner input = new Scanner(System.in);
            System.out.print("Enter employee id: ");
            int id = input.nextInt();
            System.out.print("Enter salary: ");
            double salary = input.nextDouble();
            Employee emp = new Employee(id, salary);

            System.out.printf("Client says: client passing employee to server: %s\n", emp);
            toserver.writeObject(emp);
            Object obj = fromserver.readObject();
            if (obj instanceof Employee)
                System.out.printf("Client says: client received revised employee from server: %s\n", (Employee) obj);
            else System.out.printf("Client says: Something wrong! client received back a non-employee\n");
        }

        catch(ClassNotFoundException ex)
        {
            ex.printStackTrace();
        }

        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}

class Employee implements Serializable
{
    private int id;
    private double salary;

    public Employee(int empid, double sal)
    {
        id = empid;
        salary = sal;
    }

    public void addbonus()
    {
        salary *= 1.1; // 10% bonus added
    }

    public String toString()
    {
        return "id = " + id + " and salary = " + salary;
    }
}

```

A sample session is shown below:

Output of the program – transferring objects through sockets

```

Administrator: Command Prompt
C:\Users\uday\UdayJava\network>javac ServerTransferObject.java
C:\Users\uday\UdayJava\network>java ServerTransferObject
Server says: Server started ... waiting ...
Server says: client sent this object: id = 205 and salary = 5000.0
C:\Users\uday\UdayJava\network>

Administrator: Command Prompt
C:\Users\uday\UdayJava\network>javac ClientTransferObject.java
C:\Users\uday\UdayJava\network>java ClientTransferObject
Client says: client started
Enter employee id: 205
Enter salary: 5000
Client says: client passing employee to server: id = 205 and salary = 5000.0
Client says: client received revised employee from server: id = 205 and salary = 5500.0
C:\Users\uday\UdayJava\network>

```

Module 6 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself 6.1

Assume that tables researcher, journal, and publication have been created and loaded with data. Now we wish to drop these three tables. Which of the following orders of dropping will work?

drop table researcher;
drop table publication;
drop table journal;

This is false.

drop table publication;
drop table researcher;
drop table journal;

This is true.

drop table journal;
drop table publication;
drop table researcher;

This is false.

drop table journal;
drop table researcher;
drop table publication;

This is false.

drop table researcher;
drop table journal;
drop table publication;

This is false.

```
drop table publication;  
drop table journal;  
drop table researcher;  
  
This is true.
```

Test Yourself 6.2

What, if any, is wrong with the following SQL statement?

```
Select * from journal  
group by publisher;
```

Suggested answer: This is an invalid SQL statement. The * is not an aggregate or grouping expression that is required in the selected item(s) when the group by clause is used. In other words, the * is not single-valued per group.

Test Yourself 6.3

Assume that the three insert statements in the section on SQL above have been executed to fill the three tables -- researcher, journal, and publication -- with data. Now, what would be the result of the following statement?

```
delete from journal  
where publisher = 'APS';
```

Suggested answer: The deletion attempt would fail, because the journal published by APS is currently referenced in the publication table.

Test Yourself 6.4

Given that JDBC provides the "Statement", what is the need for the "PreparedStatement"?

Suggested answer: Statement does not allow parameters, but PreparedStatement does. In other words, Statement is for static SQL, whereas PreparedStatement provides dynamic SQL.

Test Yourself 6.5

What is the difference between DatabaseMetaData and ResultSetMetaData?

Suggested answer: The DatabaseMetaData interface provides information about the database itself, while the ResultSetMetaData interface allows us to obtain information on a specific ResultSet.

Test Yourself 6.6

Does the material on networking in this module focus on TCP-based or UDP-based communication?

Suggested answer: TCP-based (stream-based) communication.

Additional Resources and References

- The Apache DB Project. *Apache Derby*. Can be retrieved from <http://db.apache.org/derby/>.
- Oracle. *Java DB Technical Documentation*. Can be retrieved from <http://docs.oracle.com/javadb>.
- Oracle. *Java DB 10.11 Documentation Overview*. Can be retrieved from <http://docs.oracle.com/javadb/support/overview.html>.
- The Apache DB Project. *Apache Derby: Documentation*. Can be retrieved from <http://db.apache.org/derby/manuals/>.
- Oracle. *Java Platform, Standard Edition (Java SE) 8*. Can be retrieved from <http://docs.oracle.com/javase/8/>.
- Oracle. *Interface ResultSet*. Can be retrieved from <http://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html>.
- Oracle. *Interface DatabaseMetaData*. Can be retrieved from <http://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html>.

