Advanced Computer Programming

[Lecture 15]

Saeed Reza Kheradpisheh

kheradpisheh@ut.ac.ir

Department of Computer Science
Shahid Beheshti University

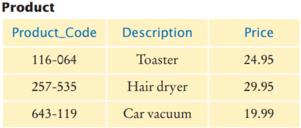
Spring 1397-98

Relational Databases

- If you have a lot of data, it can be difficult to add, remove, find, and update operations quickly and efficiently in files.
- Database management systems let the programmer think in terms of the data rather than how it is stored.
- In this chapter, you will learn how to use SQL, the Structured Query Language, to query and update information in a relational database, and how to access database information from Java programs.

Database tables

- A relational database stores information in *tables*.



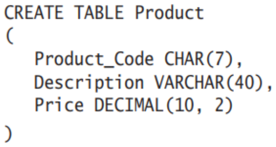
Product

Product_Code	Description	Price
116-064	Toaster	24.95
257-535	Hair dryer	29.95
643-119	Car vacuum	19.99

- Note that all items in a particular column have the same type.
- The Product table shows types that are commonly available in relational databases that follow the SQL (Structured Query Language)
- There is no relationship between SQL and Java, they are different languages.
- However, you can use Java to send SQL commands to a database.

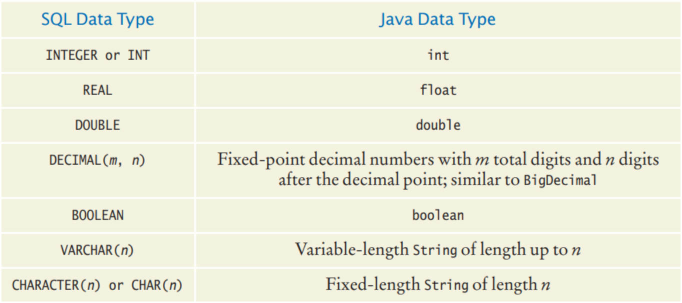
SQL: Create a table

Creat a table command:



```
CREATE TABLE Product
(
   Product_Code CHAR(7),
   Description VARCHAR(40),
   Price DECIMAL(10, 2)
)
```

SQL datatypes and the corresponding datatype in JAVA:



SQL Data Type	Java Data Type
`INTEGER or INT`	`int`
`REAL`	`float`
`DOUBLE`	`double`
`DECIMAL(`*m*`,` *n*`)`	Fixed-point decimal numbers with *m* total digits and *n* digits after the decimal point; similar to `BigDecimal`
`BOOLEAN`	`boolean`
`VARCHAR(`*n*`)`	Variable-length `String` of length up to *n*
`CHARACTER(`*n*`) or CHAR(`*n*`)`	Fixed-length `String` of length *n*

SQL: Manipulate a table

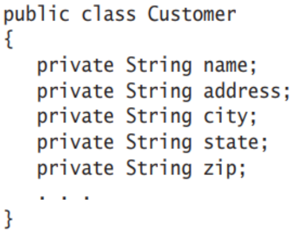
- Unlike JAVA, SQL is not case sensitive.
- For example, you could spell the command `create table` instead of `CREATE TABLE`.
- To insert rows into the table, use the `INSERT INTO` command.

  ```
  INSERT INTO Product
  VALUES ('257-535', 'Hair dryer', 29.95)
  ```
- SQL uses single quotes ('), not double quotes, to delimit strings.
- Rather than using an escape sequence (such as `\'`) as in Java, you just write the single quote twice, such as `'Sam''s Small Appliances'`
- If you create a table and subsequently want to remove it, use the `DROP TABLE` command.

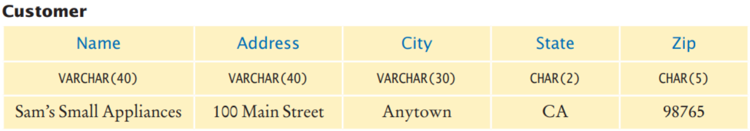
  ```
  DROP TABLE Test
  ```

Linking tables

- If you have objects whose instance variables are strings, numbers, dates, or other types that are permissible as table column types, then you can easily store them as rows in a database table.



```
public class Customer
{
   private String name;
   private String address;
   private String city;
   private String state;
   private String zip;
   . . .
}
```

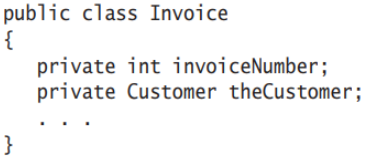


Customer

Name	Address	City	State	Zip
VARCHAR(40)	VARCHAR(40)	VARCHAR(30)	CHAR(2)	CHAR(5)
Sam's Small Appliances	100 Main Street	Anytown	CA	98765

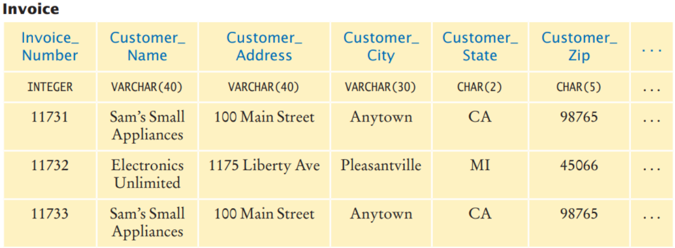
Linking tables

- For other objects (data types), it is not so easy to be stored.



```
public class Invoice
{
   private int invoiceNumber;
   private Customer theCustomer;
   . . .
}
```

- Because `Customer` isn?t a standard SQL type, you might consider simply entering all the customer data into the invoice table:



Invoice

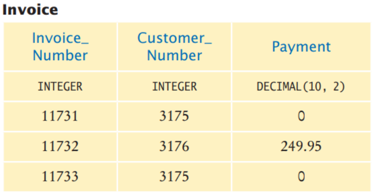
Invoice_ Number	Customer_ Name	Customer_ Address	Customer_ City	Customer_ State	Customer_ Zip	. . .
INTEGER	VARCHAR(40)	VARCHAR(40)	VARCHAR(30)	CHAR(2)	CHAR(5)	. . .
11731	Sam's Small Appliances	100 Main Street	Anytown	CA	98765	. . .
11732	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066	. . .
11733	Sam's Small Appliances	100 Main Street	Anytown	CA	98765	. . .

Linking tables

- However, this is not a good idea.
- For instance If you look at the sample data in **Invoice** table, you will notice that *Sam's Small Appliances* had two invoices, numbers 11731 and 11733.
- Yet all information for the customer was replicated in two rows.
- If the same customer places many orders, then the replicated information can take up a lot of space.
- More importantly, the replication is dangerous. Suppose the customer moves to a new address. Then it would be an easy mistake to update the customer information in some of the invoice records and leave the old address in place in others.

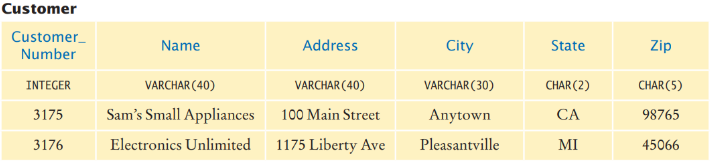
Linking tables

- The solution is to organize your data into multiple tables.



Invoice

Invoice_ Number	Customer_ Number	Payment
INTEGER	INTEGER	DECIMAL(10, 2)
11731	3175	0
11732	3176	249.95
11733	3175	0



Customer

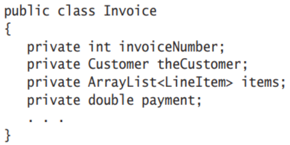
Customer_ Number	Name	Address	City	State	Zip
INTEGER	VARCHAR(40)	VARCHAR(40)	VARCHAR(30)	CHAR(2)	CHAR(5)
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

- But how can we refer to the customer to which an invoice is issued?
- Notice that there is now a `Customer_Number` column in both the **Customer** table and the **Invoice** table.
- The two tables are **linked** by the `Customer_Number` field.

Linking tables

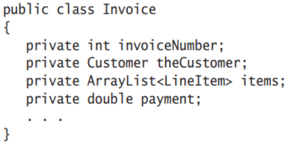
- Note that the customer number is a unique identifier.
- In database terminology, a column (or combination of columns) that uniquely identifies a row in a table is called a **primary key**.
- For example, In our *Customer table*, the `Customer_Number` column is a primary key.
- You need a primary key if you want to establish a link from another table. When a primary key is linked to another table, the matching column (or combination of columns) in that table is called a **foreign key**.
- For example, the `Customer_ Number` in the *Invoice table* is a foreign key.

Multi-Valued relationships

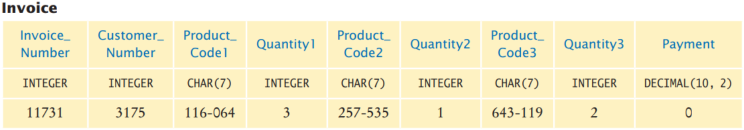


```
public class Invoice
{
   private int invoiceNumber;
   private Customer theCustomer;
   private ArrayList<LineItem> items;
   private double payment;
   . . .
}
```

Multi-Valued relationships

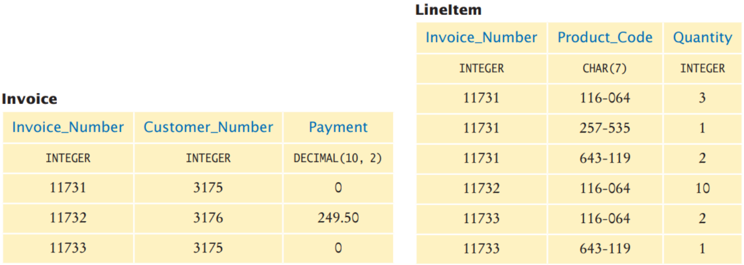


```
public class Invoice
{
   private int invoiceNumber;
   private Customer theCustomer;
   private ArrayList<LineItem> items;
   private double payment;
   . . .
}
```



Invoice

Invoice_ Number	Customer_ Number	Product_ Code1	Quantity1	Product_ Code2	Quantity2	Product_ Code3	Quantity3	Payment
INTEGER	INTEGER	CHAR(7)	INTEGER	CHAR(7)	INTEGER	CHAR(7)	INTEGER	DECIMAL(10, 2)
11731	3175	116-064	3	257-535	1	643-119	2	0



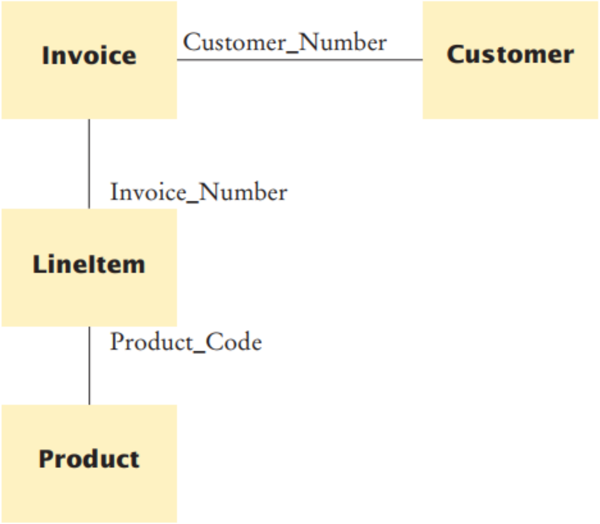
Invoice

Invoice_Number	Customer_Number	Payment
INTEGER	INTEGER	DECIMAL(10, 2)
11731	3175	0
11732	3176	249.50
11733	3175	0

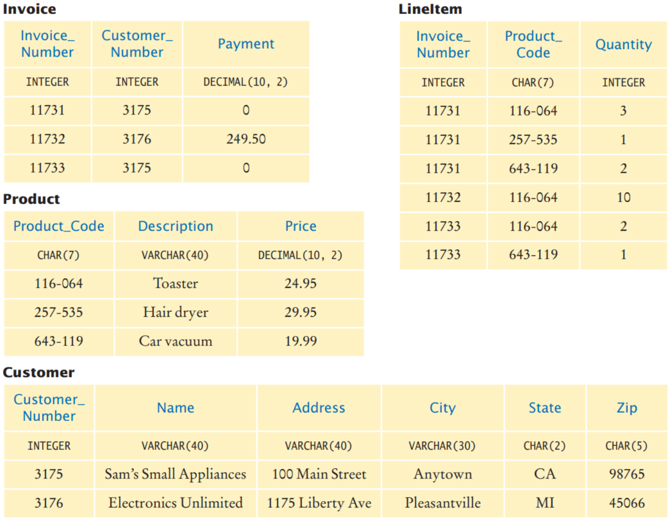
LineItem

Invoice_Number	Product_Code	Quantity
INTEGER	CHAR(7)	INTEGER
11731	116-064	3
11731	257-535	1
11731	643-119	2
11732	116-064	10
11733	116-064	2
11733	643-119	1

Multi-Valued relationships



Multi-Valued relationships



Invoice

Invoice_ Number	Customer_ Number	Payment
INTEGER	INTEGER	DECIMAL(10, 2)
11731	3175	0
11732	3176	249.50
11733	3175	0

LineItem

Invoice_ Number	Product_ Code	Quantity
INTEGER	CHAR(7)	INTEGER
11731	116-064	3
11731	257-535	1
11731	643-119	2
11732	116-064	10
11733	116-064	2
11733	643-119	1

Product

Product_Code	Description	Price
CHAR(7)	VARCHAR(40)	DECIMAL(10, 2)
116-064	Toaster	24.95
257-535	Hair dryer	29.95
643-119	Car vacuum	19.99

Customer

Customer_ Number	Name	Address	City	State	Zip
INTEGER	VARCHAR(40)	VARCHAR(40)	VARCHAR(30)	CHAR(2)	CHAR(5)
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

Queries

Once a database is filled with data, you will want to query the database for information, such as

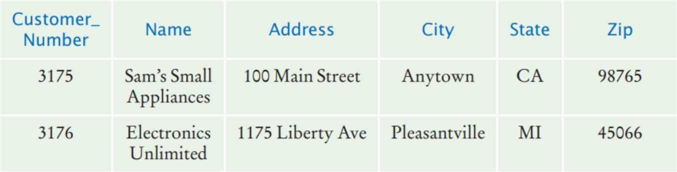
- What are the names and addresses of all customers?
- What are the names and addresses of all customers in California?
- What are the names and addresses of all customers who bought toasters?
- What are the names and addresses of all customers with unpaid invoices?

Simple Queries

- In SQL, you use the `SELECT` command to issue queries:

```
SELECT * FROM Customer
```

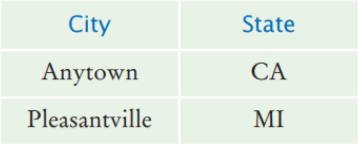
and the result is:



Customer_ Number	Name	Address	City	State	Zip
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

- Selecting columns:

```
SELECT City, State FROM Customer
```



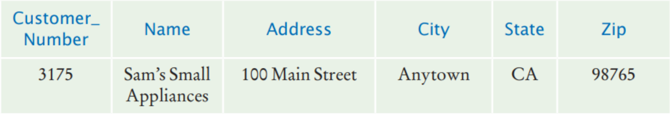
City	State
Anytown	CA
Pleasantville	MI

Simple Queries

- Selecting subsets:

  ```
  SELECT * FROM Customer WHERE State = 'CA'
  ```

  and the result is:

Customer_ Number	Name	Address	City	State	Zip
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765

- To test for inequality, you use the <> operator:

  ```
  SELECT * FROM Customer WHERE State <> 'CA'
  ```

- You can match patterns with the `LIKE` operator. The right-hand side must be a string that can contain the special symbols _ (match exactly one character) and % (match any character sequence):

  ```
  Name LIKE '_o%'
  ```

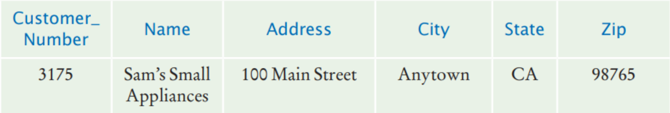
  matches all strings whose second character is an "o".

Simple Queries

- You can combine expressions with the logical connectives `AND`, `OR`, and `NOT`.

```
 SELECT *
 FROM Product
 WHERE Price < 100
 AND Description <> ’Toaster’
```

  and the result is:



Customer_ Number	Name	Address	City	State	Zip
3175	Sam’s Small Appliances	100 Main Street	Anytown	CA	98765

- Suppose you want to find out how many customers there are in California.

```
 SELECT COUNT(*) FROM Customer WHERE State = ’CA’
```

- In addition to the `COUNT` function, there are four other functions: `SUM`, `AVG (average)`, `MAX`, and `MIN`.

```
SELECT AVG(Price) FROM Product
```

Simple Queries

- Queries we have seen so far all involve a single table. However, the information we want is usually distributed over multiple tables.
- For instance, we can use a query to find the product code:

  ```
  SELECT Product_Code
  FROM Product
  WHERE Description = 'Car vacuum'
  ```

  Then we can issue a second query:

  ```
  SELECT Invoice_Number
  FROM LineItem
  WHERE Product_Code = '643-119'
  ```

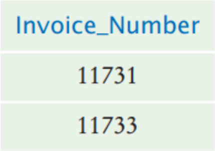
- But it makes sense to combine these two queries .

Simple Queries

- Thus, the combined query is

```
SELECT LineItem.Invoice_Number
FROM Product, LineItem
WHERE Product.Description = ’Car vacuum’
AND Product.Product_Code = LineItem.Product_Code
```

- The result is:



Invoice_Number
11731
11733

- Such a query is often called a **join** because it involves joining multiple tables.

Simple Queries

Whenever you formulate a query that involves multiple tables, remember to:

- List all tables that are involved in the query in the `FROM` clause.
- Use the `TableName.ColumnName` syntax to refer to column names.
- List all join conditions (`TableName1.ColumnName1 = TableName2.ColumnName2`) in the `WHERE` clause.

Simple Queries

Whenever you formulate a query that involves multiple tables, remember to:

- The outcome of a `SELECT` query is a result set that you can view and analyze.
- Two related statement types, `UPDATE` and `DELETE`, don't produce a result set. Instead, they modify the database.
- For instance, to delete all customers in California:

  ```
  DELETE FROM Customer WHERE State = 'CA'
  ```

- The `UPDATE` query allows you to update columns of all records that fulfill a certain condition:

  ```
  UPDATE LineItem
  SET Quantity = Quantity + 1
  WHERE Invoice_Number = '11731'
  ```

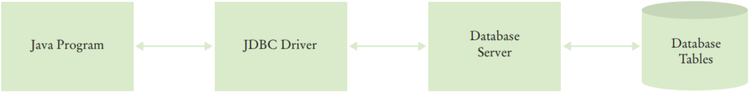
- You can update multiple column values by specifying multiple update expressions in the `SET` clause, separated by commas.

Installing a Database

A wide variety of database systems are available. Among them are

- Production-quality databases, such as Oracle, IBM DB2, Microsoft SQL Server, PostgreSQL, or MySQL
- Lightweight Java databases, such as Apache Derby, it is included with the Java Development Kit.
- Desktop databases, such as Microsoft Access.

JDBC architecture:



Database programming in Java

- To connect to a database, you need an object of the `Connection` class.
- Next, you ask the `DriverManager` for a connection.

```
String url = .  .  .;
String username = .  .  .;
String password = .  .  .;
Connection conn = DriverManager.getConnection(url,
username, password);
```

- When you are done issuing your database commands, close the database connection:

```
 conn.close();
```

Executing SQL statements

- Once you have a connection, you can use it to create `Statement` objects.

```
Statement stat = conn.createStatement();
```

- The `execute` method of the `Statement` class executes a SQL statement.

```
stat.execute("CREATE TABLE Test (Name CHAR(20))");
stat.execute("INSERT INTO Test VALUES ('Romeo')");
```

- To issue a query, use the `executeQuery` method of the `Statement` class. The query result is returned as a `ResultSet` object.

```
String query = "SELECT * FROM Test";
ResultSet result = stat.executeQuery(query);
```

Executing SQL statements

- For UPDATE statements, you can use the executeUpdate method.

  ```
  String command = "UPDATE LineItem"
  + " SET Quantity = Quantity + 1"
  + " WHERE Invoice_Number = '11731'";
  int count = stat.executeUpdate(command);
  ```

- If your statement has variable parts, then you should use a PreparedStatement instead:

  ```
  String query = "SELECT * WHERE Account_Num = ?";
  PreparedStatement stat = conn.prepareStatement(query);
  ```

- The ? symbols in the query string denote variables that you fill in when you make an actual query:

  ```
  stat.setString(1, accountNumber);
  ```

Analyzing Query Results

- The `ResultSet` class has a `next` method to visit the next row.
- The `next` method does not return any data; it returns a `boolean` value that indicates whether more data are available.
- If the result set is completely empty, then the first call to result.next() returns `false`.
- Otherwise, the first call to `result.next()` fetches the data for the first row from the database.
- Once the result set object has fetched a particular row, you can inspect its columns:

```
String productCode = result.getString("Product_Code");
int quantity = result.getInt("Quantity");
double unitPrice = result.getDouble("Price");
```

Result Set Metadata

- When you have a result set from an unknown table, you may want to know the names of the columns.
- You can use the `ResultSetMetaData` class to find out about properties of a result set:

```
 ResultSetMetaData metaData = result.getMetaData();
```

- Accessing column labels:

```
for (int i = 1; i <= metaData.getColumnCount(); i++)
{
String columnName = metaData.getColumnLabel(i);
int columnSize = metaData.getColumnDisplaySize(i);
...
}
```