Experiment No. 08

Title : Program for creation of Frame and Window using Swing.

Aim : To Study Frame, AWT Controls and create an application using Frame, AWT Controls

Objective : Implement a program to understand the concept of Frame and AWT controls.

Theory : **ABSTRACT WINDOW TOOLKIT** (AWT)

The **Abstract Window Toolkit** (AWT) is Java's original platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is now part of the Java Foundation Classes (JFC) the standard API for providing a graphical user interface (GUI) for a Java program.

AWT is also the GUI toolkit for a number of Java ME profiles. For example, Connected Device Configuration profiles require Java runtimes on mobile telephones to support AWT.The AWT contains numerous classes and methods that allow you to create and manage windows

AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java's largest

packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table below lists some of the many AWT classes.

Table: Some AWT Classes

Class	Description
Checkbox	**Creates a check box control.**
CheckboxMenuItem	**Creates an on/off menu item.**
Component	**An abstract superclass for various AWT components.**
Frame	**Creates a standard window that has a title bar, resize corners, and a menu bar.**
Label	**Creates a label that displays a string.**
List	**Creates a list from which the user can choose. Similar to the standard Windows list box.**
Menu	**Creates a pull-down menu.**
MenuBar	**Creates a menu bar.**
MenuComponent	**An abstract class implemented by various menu classes.**
MenuItem	**Creates a menu item.**

Window	**Creates a window with no frame, no menu bar, and no title.**
AWTEvent	**Encapsulates AWT Event**
Button	**Creates push button**
AWTEventMulticaster	**Dispatches event to multiple listeners**
BorderLayout	**Supports for five components:North, South,East,West,Center**

Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel**, which is used by applets, and those derived from **Frame**, which creates a standard window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 1.shows the class hierarchy for **Panel** and **Frame**. Let's look at each of these classes now.

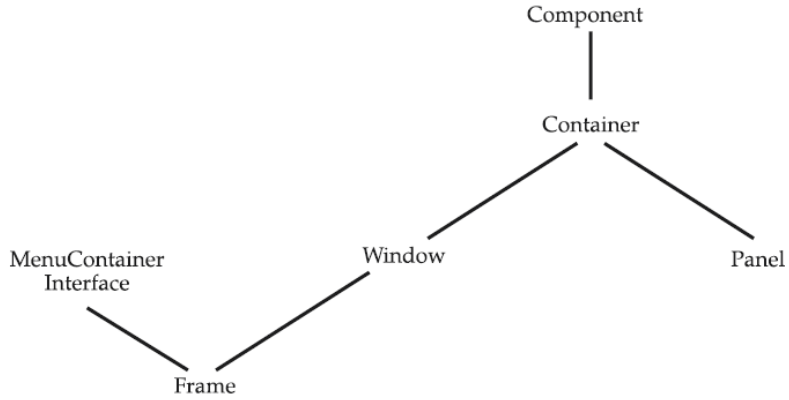


Figure 1: The class hierarchy for Panel and Frame

Component:

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. (You already used many of these methods when you created applets in Chapters 19 and 20.) A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

The **Container** class is a subclass of **Component**. It has additional methods that allow

other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains.

Panel

The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, or **setBounds( )** methods defined by **Component**.

Window

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. If you create a **Frame** object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user's knowledge.) When a **Frame** window is created by a program rather than an applet, a normal window is created.

Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. **Canvas** encapsulates a blank window upon which you can draw.

Working with Frame Windows

After the applet, the type of window you will most often create is derived from **Frame**. You will use it to create child windows within applets, and top-level or child

windows for applications. As mentioned, it creates a standard-style window. Here are two of **Frame**'s constructors:
Frame( )
Frame(String *title*)

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by *title.* Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created.

There are several methods you will use when working with **Frame** windows. They are examined here. Setting the Window's Dimensions The **setSize( )** method is used to set the dimensions of the window. Its signature is shown here:
void setSize(int *newWidth*, int *newHeight*)
void setSize(Dimension *newSize*)

The new size of the window is specified by *newWidth* and *newHeight*, or by the **width** and **height** fields of the **Dimension** object passed in *newSize*. The dimensions are specified in terms of pixels. The **getSize( )** method is used to obtain the current size of a window. Its signature
is shown here:
Dimension getSize( )
This method returns the current size of the window contained within the **width** and **height** fields of a **Dimension** object. Hiding and Showing a Window After a frame window has been created, it will not be visible until you call **setVisible( )**. Its signature is shown here:
void setVisible(boolean *visibleFlag*)

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.
Setting a Window's Title You can change the title in a frame window using **setTitle( )**, which has this general form:
void setTitle(String *newTitle*)
Here, *newTitle* is the new title for the window.

Closing a Frame Window
When using a frame window, your program must remove that window from the screen when it is closed, by calling **setVisible(false)**. To intercept a window-close event, you must implement the **windowClosing( )** method of the **WindowListener** interface. Inside **windowClosing( )**, you must remove the window from the screen. The example in the next section illustrates this technique.
Creating a Frame Window in an Applet

While it is possible to simply create a window by creating an instance of **Frame**, you will seldom do so, because you will not be able to do much with it. For example, you will not be able to receive or process events that occur within it or easily output information to it. Most of the time, you will create a subclass of **Frame**. Doing so lets you override **Frame**'s methods and event handling.

Creating a new frame window from within an applet is actually quite easy. First, create
a subclass of **Frame**. Next, override any of the standard window methods, such as **init( )**, **start( )**, **stop( )**, and **paint( )**. Finally, implement the **windowClosing( )** method of the **WindowListener** interface, calling **setVisible(false)** when the window is closed. Once you have defined a **Frame** subclass, you can create an object of that class. This causes a frame window to come into existence, but it will not be initially visible. You make it visible by calling **setVisible( )**. When created, the window is given a default height and width. You can set the size of the window explicitly by calling the **setSize( )** method.

The following applet creates a subclass of **Frame** called **SampleFrame**. A window of this subclass is instantiated within the **init( )** method of **AppletFrame**. Notice that **SampleFrame** calls **Frame**'s constructor. This causes a standard frame window to be created with the title passed in **title**. This example overrides the applet window's **start( )** and **stop( )** methods so that they show and hide the child window, respectively. This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page. It also causes the child window to be shown when the browser returns to the applet.

JFC and Swing

JFC is short form of Java Foundation Classes, which encompass a group of features for building graphical user interfaces (GUIs) and adding rich graphics functionality and interactivity to Java applications. It is defined as containing the features shown in the table below.

Features of the Java Foundation Classes	
Feature	**Description**
Swing GUI Components	Includes everything from buttons to split panes to tables. Many components are capable of sorting, printing, and drag and drop, to name a few of the supported features.
Pluggable Look-and-Feel Support	The look and feel of Swing applications is pluggable, allowing a choice of look and feel. For example, the same program can use either the Java or the Windows look and feel. Additionally, the Java platform supports the GTK+ look and feel, which makes hundreds of existing look and feels

	available to Swing programs. Many more look-and-feel packages are available from various sources.
Accessibility API	Enables assistive technologies, such as screen readers and Braille displays, to get information from the user interface.
Java 2D API	Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.
Internationalization	Allows developers to build applications that can interact with users worldwide in their own languages and cultural conventions. With the input method framework developers can build applications that accept text in languages that use thousands of different characters, such as Japanese, Chinese, or Korean.

This trail concentrates on the Swing components. We help you choose the appropriate components for your GUI, tell you how to use them, and give you the background information you need to use them effectively. We also discuss other JFC features as they apply to Swing components.

Which Swing Packages Should I Use?

The Swing API is powerful, flexible — and immense. The Swing API has 18 public packages:

```
javax.accessibility      javax.swing.plaf       javax.swing.text
javax.swing              javax.swing.plaf.basic javax.swing.text.html
javax.swing.border       javax.swing.plaf.metal javax.swing.text.html.parser
javax.swing.colorchooser javax.swing.plaf.multi javax.swing.text.rtf
javax.swing.event        javax.swing.plaf.synth javax.swing.tree
javax.swing.filechooser  javax.swing.table      javax.swing.undo
```

Fortunately, most programs use only a small subset of the API. This trail sorts out the API for you, giving you examples of common code and pointing you to methods and classes you're likely to need. Most of the code in this trail uses only one or two Swing packages:

- `javax.swing`
- `javax.swing.event` (not always required)

Program:

Output:

Conclusion Thus we have studied about program creation of frame and window using swing.

Experiment No. 9

Title : Program to demonstrate Event handling concept.

Aim : To study Graphical user interface design in Java

Objective : Event-based applications are highly used in GUI, which is essential part of the application development

Theory : Java, while only a few years old, is already being deployed in a wide variety of devices. Java exists for mainframes, midrange servers, PCs, and handheld devices. There is even a group currently working on a real-time Java for embedded control applications! Aside from the embedded environment, the vast majority of Java developers will be required to create some type of user interface for their application. For things like configuration tools, a command-line interface works fine (which Java supports, of course). However, for applications deployed to a wide range of users or for applications required to display data, graphical user interfaces work best.

Java supports GUI development through the Abstract Windowing Toolkit, or AWT. The AWT is the Java equivalent of the Microsoft Windows Common Control Library or a Motif widget toolkit. It includes support for simple graphics programming as well as a number of preconstructed components such as button, menu, list, and checkbox classes. The java.awt package is included with the Java 2 SDK and will be the focus of this discussion. Numerous third-party components are available (see links at bottom of this page) for additional GUI functionality.

Java AWT Basics

The AWT allows Java developers to quickly build Java applets and applications using a group of prebuilt user interface components. A number of Java IDE's are available which support the creation of user interfaces using the AWT by dragging-and-dropping components off a toolbar. It should be noted that these IDE's actually generate Java code on the fly based on what you do in the graphical design environment. This is in contrast to toolsets such as Microsoft Visual Basic which separate user interface design from the application code. The advantage of the Java approach is that you can edit your GUI either through a graphical IDE or by simply modifying the Java code and recompiling.

The three steps common to all Java GUI applications are:

1. Creation of a container
2. Layout of GUI components
3. Handling of events.

1. Creating A Container

This container object is actually derived from the java.awt.Container class and is one of (or inherited from) three primary classes: java.awt.Window, java.awt.Panel, java.awt.ScrollPane. The Window class represents a standalone window (either an application window in the form of a java.awt.Frame, or a dialog box in the form of a java.awt.Dialog).

The java.awt.Panel class is not a standalone window in and of itself; instead, it acts as a background container for all other components on a form. For instance, the java.awt.Applet class is a direct descendant of java.awt.Panel.

Laying Out GUI Components

GUI components can be arranged on a container using one of two methods. The first method involves the exact positioning (by pixel) of components on the container. The second method involves the use of what Java calls Layout Managers. If you think about it, virtually all GUIs arrange components based on the row-column metaphor. In other words, buttons, text boxes, and lists generally aren't located at random all over a form. Instead, they are usually neatly arranged in rows or columns with an OK or Cancel button arranged at the bottom or on the side. Layout Managers allow you to quickly add components to the manager and then have it arrange them automatically. The AWT provides six layout managers for your use:

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.CardLayout
- java.awt.GridLayout
- java.awt.GridBagLayout
- java.awt.BoxLayout

The Container class contains the setLayout() method so that you can set the default LayoutManager to be used by your GUI. To actually add components to the container, we use the container's add() method:

```
Panel p = new java.awt.Panel();
Button b = new java.awt.Button("OK");
p.add(b);
```

Event Handling in Java 2

- Events are processed by listener objects which are registered by the GUI elements.

- In order to be used as a listener, a class must implement a listener interface.
- When an event occurs, the object on which the event occurred (source) sends the event to all registered listeners interested in that event by invoking the appropriate method on the listener object. An EventObject object is passed as an argument to the method. The Listener object can then take action based on that event or can ignore it.
- You do not have to specify separate listener classes -- the Applet itself can implement its own listeners.
- Or the button subclass could implement the listener interface
- There are several types of listener objects - each handles actions from several GUI element types. From Java Examples in a Nutshell:

component	Events generated	Meaning
Button	ActionEvent	User clicked on button
Checkbox	ItemEvent	User selected or deselected item
CheckboxMenuItem	ItemEvent	User selected or deselected item
Choice	ItemEvent	User selected or deselected item
Component	ComponentEvent	Component moved hidden ,resized

- For a class to be a listener, it must implement one or more of the Listener interfaces: ActionListener, AdjustmentListener, ComponentListener,FocusListener, ItemListener, KeyListener, MouseListener, MouseMotionListener, TextListener, WindowListener. See here for the functions which are associated with each.
- A listener must implement all the methods in the listener interface. However, they can have empty bodies ({}) if not of interest.
- As listed above, the EventObject has subclasses (such as MouseEvent) that have their own methods. So, for example, a MouseEvent has getX() and getY() to get the coordinates of the mouse event (relative to the source component)).

Program:

Output:

Conclusion Thus we studied how to handle event generated from AWT controls.

Experiment No. 10

Title : Implementing the concept of Network Programming

Aim : To study

1. Networking basics
2. Socket Programming

Objective : The classes of java.net packages internally use TCP/IP and UDP protocol that are responsible for sending and receiving data. We are also establish communication between a server and a client by creating server socket and client socket.

Theory :

Java Networking

Network access is crucial to computer operations in the twenty first century. Java provides many built-in networking class objects through its .net, .nio and .rmi packages. java.net provides http connections and streams as well as protocol sockets. java.nio, java.nio.charset and java.nio.channels provide buffers, character sets and channels for multiplexed non-blocking applications that are more tolerant of dropped connections and time delays. java.rmi provides methods for remote method invocation. Internet addresses uniquely identify each computer on the network. They use a 4 byte dot IP notation such as 220.210.34.7 Domain Name Service allows us to use an easier to remember naming scheme. DNS servers translate the domain name such as amazon.com into its dot IP address.

Servers are any computer with resources (such as printers and disks) to be shared. Clients are entities that want to use these resources. Servers listen to their socket ports waiting for a client to connect with a service request. Servers are multithreaded to permit multiple services and multiple connects to the same service simultaneously. Proxy servers speak the client side of the protocol to another server. It acts as an agent of the client and can be set up to filter or cache data for it.

A network socket is a standard interface that uses the TCP/IP protocol to interconnect with a network. Sockets use IP address extensions or ports to connect to specific services such as FTP and telnet. Each port number (below 1024) has a reserved use.

Internet Streams

Internet streams allow you to access remote document data. The java.net package provides several objects for networking at the stream level:

- URL - builds a valid internet address. Methods are: getFile(), getHost(), getPort(), getProtocol(), openConnection() and toExternalForm()

- URLConnection - is a general purpose connection. Methods are: connect(), getContent(), getContentLength(), getContentType(), getDate(), getExpiration(), getInputStream() and getLastModified()
- HttpURLConnection - is an Http protocol specific connection. Methods in addition to those inherited from URLConnection are: getHeaderField(), getHeaderFieldDate(), getHeaderFieldKey(), getRequestMethod(), setRequestMethod(), [unfinished]

GetFile.java (found in jp7net.zip) demonstrates how to create a utility that displays both the header and the contents of an internet document. Seven main steps are involved:

1. Create an URL object that represents the resource's WWW address.
2. Create a HttpURLConnection object that opens a connection for the URL.
3. Use the getContent() method to create an InputStreamReader.
4. Use the InputStreamReader to create a BufferedReader object.
5. Use the getHeaderFieldKey(idx) and getHeaderField(idx) methods to retrieve header information.
6. Read the contents from the BufferedReader stream.
7. Write the contents to a Swing GUI textbox.

Warning: Both the access and display methods do not work for media files! If you are fetching image or audio files, use the technique from the packager project.

Socket Programming

TCP/IP sockets can be used to access protocols other than http. The java.net package provides several objects for networking at the TCP/IP socket level:

- Socket - TCP/IP client socket for telnet, smtp, nntp, whois, finger etc.
- ServerSocket - TCP/IP socket for server-side applications.

The following programs duplicate existing client utilities but demonstrate how to write TCP/IP applications.

Whois.java (found in jp7net.zip) accesses a directory service that provides information about a specific host. It makes a socket connection to internic.net (the registrar for commercial sites), sets a timeout and establishes a stream to access the data. Once compiled, test Whois with **java Whois amazon.com**.

Finger.java (found in jp7net.zip) accesses a directory service that provides information about a particular user based on his .plan and .project files. It makes a socket connection, sets a timeout and establishes a stream to access the data.

Unfortunately most hosts have removed their finger software because of address harvesting. One remaining site for testing is hlr@well.com (aka hwl@well.sf.ca.us).

TimeServer.java (found in jp7net.zip) is a ServerSocket application that places the current time on its port 4415. Any client can access it. To test the application start the server with **java TimeServer**. A window should open with the message TimServer running... Do not close the window. On XP machines use the RUN dialog **telnet localhost 4415**. On older platforms start telnet in another window. At the connect dialog enter localhost in the Host Name field and 4415 in the Port field

Program:

Output:

Conclusion Thus we studied how to use the networking concept for socket programming using Java .

Experiment No. 11

Title : Program to demonstrate various methods of Collection class.

Aim : To Study the Collection class and its methods.

Objective : Implement a program to understand the concept of Collection class and its methods.

Theory :

Collections in Java :

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (Array List, Vector, Linked List, Priority Queue, Hash Set, Linked Hash Set, Tree Set etc).

What is Collection in java

Collection represents a single unit of objects i.e. a group.

What is framework in java

- provides readymade architecture.
- represents set of classes and interface.
- is optional.

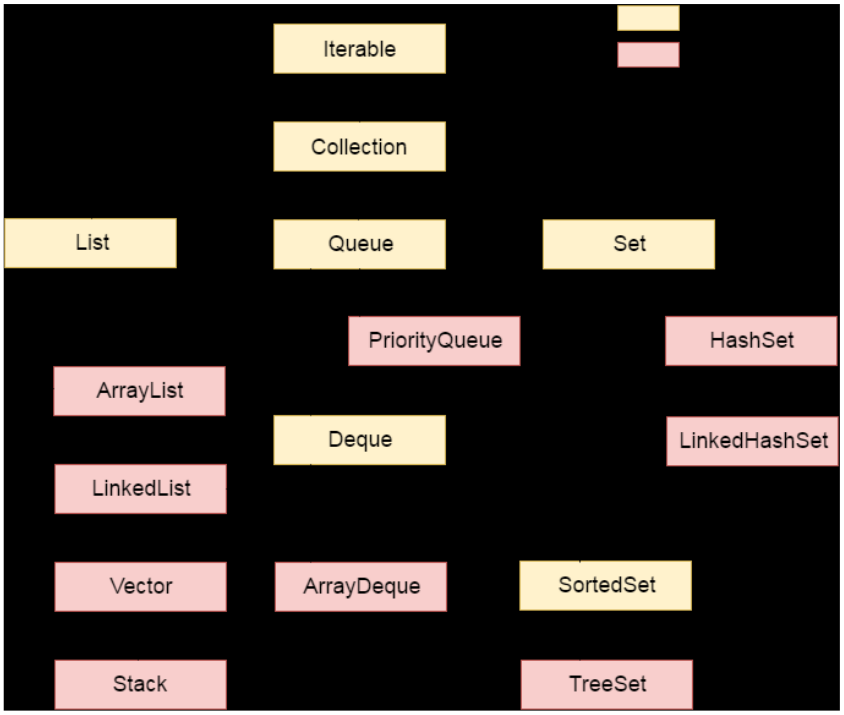
What is Collection framework

Collection framework represents a unified architecture for storing and manipulating group of objects. It has:

1. Interfaces and its implementations i.e. classes
2. Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of collection framework. The **java.util** package contains all the classes and interfaces for Collection framework.



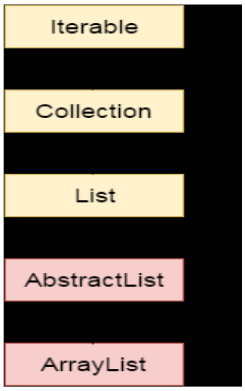
Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Hierarchy of ArrayList class



Methods of Java ArrayList

- Void add()- It is used to insert the specified element at a specified position index in a list.

- Void clear() –It is used to remove all elements of the list.

Two ways to iterate the elements of collection in java

There are two ways to traverse collection elements:
1. By Iterator interface.

2. By for-each loop.

Program:

Output:

Conclusion Thus we studied the Collection class and its various methods.

Experiment No. 12

Title : Program for Database Connectivity using JDBC and ODBC

Aim : To study JDBC and ODBC connectivity in Java.

Objective : Database plays an important role in various applications. JDBC helps in communicate with any database through a java program which is an essential feature for any project development environment.

Theory :

JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC helps you to write java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

   The following simple code fragment gives a simple example of these three steps:

```
4. Connection con = DriverManager.getConnection
   ("jdbc:myDriver:wombat", "myLogin","myPassword");
5. Statement stmt = con.createStatement();
6. ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
7. while (rs.next()) {
8.     int x = rs.getInt("a");
9.     String s = rs.getString("b");
10.    float f = rs.getFloat("c");
11. }
```

   This short code fragment instantiates a `DriverManager` object to connect to a database driver and log into the database, instantiates a `Statement` object that carries your SQL language query to the database; instantiates a `ResultSet` object that retrieves the results of your query, and executes a simple `while` loop, which retrieves and displays those results. It's that simple.

JDBC Product Components

JDBC includes four components:

1. The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

   The JDBC API is part of the Java platform, which includes the Java™ Standard Edition (Java™ SE) and the Java™ Enterprise Edition (Java™ EE). The JDBC 4.0 API is divided into two packages: `java.sql` and `javax.sql`. Both packages are included in the Java SE and Java EE platforms.
2. **JDBC Driver Manager —**

   The JDBC `DriverManager` class defines objects which can connect Java applications to a JDBC driver. `DriverManager` has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

   The Standard Extension packages `javax.naming` and `javax.sql` let you use a `DataSource` object registered with a Java Naming and Directory Interface™ (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism, but using a `DataSource` object is recommended whenever possible.
3. **JDBC Test Suite**

   The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.
4. **JDBC-ODBC Bridge —**

   The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

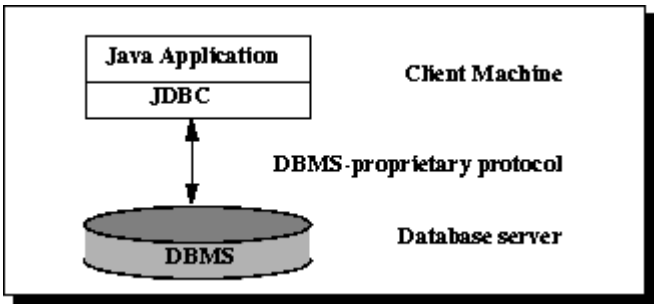
This Trail uses the first two of these four JDBC components to connect to a database and then build a java program that uses SQL commands to communicate with a test Relational Database. The last two components are used in specialized

environments to test web applications, or to communicate with ODBC-aware DBMSs.

Two-tier and Three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

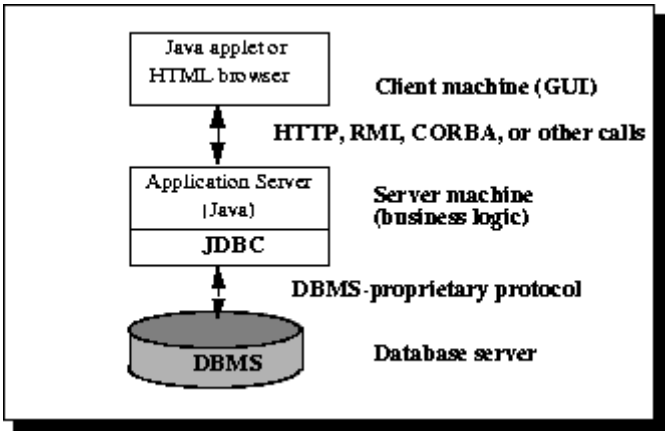
Figure 1: Two-tier Architecture for Data Access.



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Figure 2: Three-tier Architecture for Data Access.



Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

A Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table

should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a null value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a primary key. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

Table 1.2 illustrates some of these relational database concepts. It has five columns and six rows, with each row representing a different employee.

Table 1.2: Employees

Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	Axel	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

The primary key for this table would generally be the employee number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also be possible to use First_Name and Last_Name because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of "Washington." In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If Elizabeth Taylor gets a job at this company and the primary key is First_Name, the RDBMS will not allow her name to be added (if it has been specified that no duplicates are permitted). Because there is already an Elizabeth in the table, adding a second one would make

the primary key useless as a way of identifying just one row. Note that although using First_Name and Last_Name is a unique composite key for this example, it might not be unique in a larger database. Note also that Table 1.2 assumes that there can be only one car per employee.

SELECT Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the SELECT statement.

A SELECT statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements. A SELECT statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the Car_Number column) follows. The first name and last name are printed for each row that satisfies the requirement because the SELECT statement (the first line) specifies the columns First_Name and Last_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

```
FIRST_NAME          LAST_NAME
----------          -----------
Axel                Washington
Florence            Wojokowski
```

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no `WHERE` clause). Note that `SELECT *` means "SELECT all columns."

```
SELECT *
FROM Employees
```

`WHERE` Clauses

The `WHERE` clause in a `SELECT` statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if

they occur in a row in which the column Last_Name begins with the string 'Washington'.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

The keyword `LIKE` is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Washington', which signifies that any value containing the string 'Washington' plus zero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in `LIKE` clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.

The code fragment below has a `WHERE` clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the employee who is assigned car 12.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number > 10005
```

`WHERE` clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated `WHERE` clauses, but the following code fragment has a `WHERE` clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100 and Car_Number IS NULL
```

A special type of `WHERE` clause involves a join, which is explained in the next section.

Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car. This information is stored in another table, Cars, shown in Table 1.3.

Table 1.3. Cars

Car Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is Car_Number, which is the primary key for the table Cars and the foreign key in the table Employees. If the 1996 Honda Civic were wrecked and deleted from the Cars table, then Car_Number 5 would also have to be removed from the Employees table in order to maintain what is called referential integrity. Otherwise, the foreign key column (Car_Number) in Employees would contain an entry that did not refer to anything in Cars. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the Car_Number column in the table Employees because it is possible for an employee not to have a company car.

The following code asks for the first and last names of employees who have company cars and for the make, model, and year of those cars. Note that the `FROM` clause lists both Employees and Cars because the requested data is contained in both tables. Using the table name and a dot (.) before the column name indicates which table contains the column.

```
SELECT Employees.First_Name, Employees.Last_Name, Cars.Make,
      Cars.Model, Cars.Year
FROM Employees, Cars
WHERE Employees.Car_Number = Cars.Car_Number
```

This returns a result set that will look similar to the following:

```
FIRST_NAME   LAST_NAME     MAKE       MODEL     YEAR
```

```
---------- ---------- -------- --------- -------
Axel       Washington Honda    CivicDX   1996
Florence   Wojokowski Toyota   Corolla   1999
```

Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.

A list of the more common DML commands follows:

- `SELECT` – used to query and display data from a database. The `SELECT` statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are `SELECT` statements.
- `INSERT` – adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- `DELETE` – removes a specified row or set of rows from a table
- `UPDATE` – changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- `CREATE TABLE` – creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. `CREATE TABLE` is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
- `DROP TABLE` – deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TABLE command as specified by SQL92, Transitional Level. However, support for the `CASCADE` and `RESTRICT` options of `DROP TABLE` is optional. In addition, the behavior of `DROP TABLE` is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.
- `ALTER TABLE` – adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

The following code is an example of how to create a very simple stored procedure using the Java programming language. Note that the stored procedure is just a static

Java method that contains normal JDBC code. It accepts two input parameters and uses them to change an employee's car number.

Do not worry if you do not understand the example at this point. The code example below is presented only to illustrate what a stored procedure looks like. You will learn how to write the code in this example in the tutorials that follow.

```
import java.sql.*;

public class UpdateCar {

   public static void UpdateCarNum(int carNo, int empNo)
                                 throws SQLException {
      Connection con = null;
      PreparedStatement pstmt = null;

      try {
        con = DriverManager.getConnection("jdbc:default:connection");

         pstmt = con.prepareStatement(
                  "UPDATE EMPLOYEES SET CAR_NUMBER = ? " +
                  "WHERE EMPLOYEE_NUMBER = ?");
         pstmt.setInt(1, carNo);
         pstmt.setInt(2, empNo);
         pstmt.executeUpdate();
      }
      finally {
         if (pstmt != null) pstmt.close();
      }
   }
}
```

Metadata

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface `DatabaseMetaData`, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata. In general,

developers writing tools and drivers are the ones most likely to be concerned with metadata.

Establishing a Connection

First, you need to establish a connection with the DBMS you want to use. Typically, a JDBC™ application connects to a target data source using one of two mechanisms:

- `DriverManager`: This fully implemented class requires an application to load a specific driver, using a hardcoded URL. As part of its initialization, the `DriverManager` class attempts to load the driver classes referenced in the `jdbc.drivers` system property. This allows you to customize the JDBC Drivers used by your applications.
- `DataSource`: This interface is preferred over `DriverManager` because it allows details about the underlying data source to be transparent to your application. A `DataSource` object's properties are set so that it represents a particular data source.

Establishing a connection involves two steps: Loading the driver, and making the connection.

Loading the Driver

Loading the driver you want to use is very simple. It involves just one line of code in your program. To use the Java DB driver, add the following line of code:

Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

Your driver documentation provides the class name to use. In the example above, `EmbeddedDriver` is one of the drivers for Java DB.

Calling the `Class.forName` automatically creates an instance of a driver and registers it with the `DriverManager`, so you don't need to create an instance of the class. If you were to create your own instance, you would be creating an unnecessary duplicate, but it would do no harm.

After you have loaded a driver, it can make a connection with a DBMS.

Making the Connection

The second step in establishing a connection is to have the appropriate driver connect to the DBMS.

Using the DriverManager Class

The `DriverManager` class works with the `Driver` interface to manage the set of drivers available to a JDBC client. When the client requests a connection and provides a URL, the `DriverManager` is responsible for finding a driver that recognizes the URL and for using it to connect to the corresponding data source. Connection URLs have the following form:

```
jdbc:derby:<dbName>[propertyList]
```

The `dbName` portion of the URL identifies a specific database. A database can be in one of many locations: in the current working directory, on the classpath, in a JAR file, in a specific Java DB database home directory, or in an absolute location on your file system.

If you are using a vendor-specific driver, such as Oracle, the documentation will tell you what subprotocol to use, that is, what to put after `jdbc:` in the JDBC URL. For example, if the driver developer has registered the name `OracleDriver` as the subprotocol, the first and second parts of the JDBC URL will be `jdbc.driver.OracleDriver` . The driver documentation will also give you guidelines for the rest of the JDBC URL. This last part of the JDBC URL supplies information for identifying the data source.

The `getConnection` method establishes a connection:

```
Connection conn = DriverManager.getConnection("jdbc:derby:COFFEES");
```

In place of " `myLogin` " you insert the name you use to log in to the DBMS; in place of " `myPassword` " you insert your password for the DBMS. So, if you log in to your DBMS with a login name of " `Fernanda` " and a password of " `J8,` " just these two lines of code will establish a connection:

```
String url = "jdbc:derby:Fred";
Connection con = DriverManager.getConnection(url, "Fernanda", "J8");
```

If one of the drivers you loaded recognizes the JDBC URL supplied to the method `DriverManager.getConnection`, that driver establishes a connection to the DBMS specified in the JDBC URL. The `DriverManager` class, true to its name, manages all of the details of establishing the connection for you behind the scenes. Unless you are writing a driver, you probably won't use any of the methods in the interface `Driver`, and the only `DriverManager` method you really need to know is `DriverManager.getConnection`

The connection returned by the method `DriverManager.getConnection` is an open connection you can use to create JDBC statements that pass your SQL statements to

the DBMS. In the previous example, `con` is an open connection, and you use it in the examples that follow.

Using a DataSource Object for a connection

Using a `DataSource` object increases application portability by making it possible for an application to use a logical name for a data source instead of having to supply information specific to a particular driver. The following example shows how to use a `DataSource` to establish a connection:

You can configure a `DataSource` using a tool or manually. For example, Here is an example of a `DataSource` lookup:

```
InitialContext ic = new InitialContext()

DataSource ds = ic.lookup("java:comp/env/jdbc/myDB");
Connection con = ds.getConnection();
DataSource ds = (DataSource) org.apache.derby.jdbc.ClientDataSource()
ds.setPort(1527);
ds.setHost("localhost");
ds.setUser("APP")
ds.setPassword("APP");

Connection con = ds.getConnection();
```

`DataSource` implementations must provide getter and setter methods for each property they support. These properties typically are initialized when the `DataSource` object is deployed.

```
VendorDataSource vds = new VendorDataSource();
vds.setServerName("my_database_server");
String name = vds.getServerName();
```

Program:

Output:

Conclusion Thus we studied Program for Database Connectivity using JDBC and ODBC