ADVANCE PROGRAMMING

COURSE TUTORIAL

Contents

Course Objectives

Upon completion of this course, students should be able to:

- Use Java technology data types and expressions
- Use Java technology flow control constructs
- Use arrays and other data collections
- Error-handling techniques using exception handling
- GUI using Java technology GUI components: panels, buttons, labels, text fields, and text areas
- Multithreaded programs
- Java Database Connectivity (JDBC)

Chapter One: Overview of Java Programming

1.1 Creation of Java

In 1991, Sun Microsystems develop a new language. It was named Java in 1995. Java is a platform-independent language (architecture neutral). The platform-independent language could be used to produce code that would run on a variety of CPUs under differing environments. Programs are inbuilt with the internet for active programs.

Java Characteristics

- Java is **simple** in that it retains much familiar syntax of C++.
- **It is strongly typed.** This means that everything must have a data type.
- Java performs its own memory management avoiding the memory leaks that plague programs written in languages like C and C++.
- Java is completely **object-oriented**.
- Java is **multi-threaded**, allowing a program to do more than one thing at a time.
- **Network-Savvy**: an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP.
- **Secure** and **Robust**: Java is intended for writing programs that are reliable and secure in a variety of ways.
- **Portable** and **architecture neutral**.

1.2 Data types and Variables

A. Data Types

In Java, there are two categories of data types:

- **Primitive (Basic/Simple) types**: These consist of int, float, char, and Boolean.
- **Reference types**: It consists of interfaces, classes, and arrays

B. Variables

- It is the basic unit of storage in a Java program.
- It is defined by the combination of an identifier, a type, and an optional initializer.

Declaring Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [= value][, identifier [= value] ...] ;

The type is one of Java's atomic types or the name of a class or interface. The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value is called a constant variable. **For Example:**

```
int a, b, c;     // declares three integers a, b, and c.
int d = 3, e, f = 5;   //constant variable
double pi = 3.14159;  // declares an approximation of pi.
char x = 'x';
```

Dynamic declaration:

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
double  c = Math.sqrt(a * a + b * b);
```

1.3 Selection and Loop statement

I. Selection Statement

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

a. <u>If statement</u>: A simple if statement executes an action if and only if the condition is true.

Syntax:
```
if(boolean-expression)
    statement;
```

b. <u>If…else statement</u>: To take alternative actions when the condition is false, you can use an if…else statement.

Syntax:
```
if(boolean-expression)
        statement for true case;
else
        statement for false case;
```

c. <u>Nested if</u>: A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement.

```
if(i == 10) {
        if(j < 20) a = b;
        if(k > 100)
```

```
                    c = d;
            else
                    a = c;
    }else
            a = d;
```

d. <u>if-else-if Ladder</u>

```
    if(condition)
            statement;
    else if(condition)
            statement;
    else if(condition)
            statement;
    .
    .
    else
            statement;
```

e. <u>Switch</u>: The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements.

Syntax:

```
switch (switch-expression){
        case value1: statement(s)1;
            break;
        case value2: statement(s)2;
            break;
        ...
        case valueN: statement(s)N;
            break;
        default:    statement(s)-for-default;
}
```

II. Loop Statement

Java's iteration statements are for, while, and do-while. A loop repeatedly executes the same set of instructions until a termination condition is met.

a. <u>While Loop</u>

Syntax:
```
while (condition){
        // body of the loop
        Statement(s);
}
```

The body of the loop will be executed as long as the conditional expression is true. When the condition becomes false, control passes to the next line of code immediately following the loop.

b. <u>do…while Loop</u>: This loop always execute its body at least once, because its conditional expression is at the bottom of the loop.

Syntax:
```
do{
        // body of the loop
} while (condition);
```

c. <u>For</u>: For loop is a powerful and versatile construct.

Syntax:
```
for (initialization; condition; iteration) {
        // body of the loop
}
```

It is important to understand that the initialization expression is only executed once. Next, the condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.

1.4 Methods and Arrays

I. Methods

Syntax:
```
modifier returnType methodName(parameters){
        // method body;
}
```

In Java, it is possible to define two or more methods within the same class that shares the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java implements polymorphism.

For Example void test ()

void test (int a)

void test (int a, int b)

double test (double a)

II. Arrays

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.

a. One-dimensional array

Syntax: type var-name []; // Eg. double myList [];

The value of myList is set to null, which represents an array with no value. To link myList with an actual, physical array of integers, you must allocate one using new and assign it to myList. **new** is a special operator that allocates memory.

array-var = new type [size];

myList = new double [10];

b. Multidimensional array

In Java, multidimensional arrays are arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. It is used to represent a table or matrix.

For example: int twoD[][] = new int [4][5];

1.5 Java Exception Handling

What is "Exception"?

An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Opening an unavailable file
- Code errors etc.

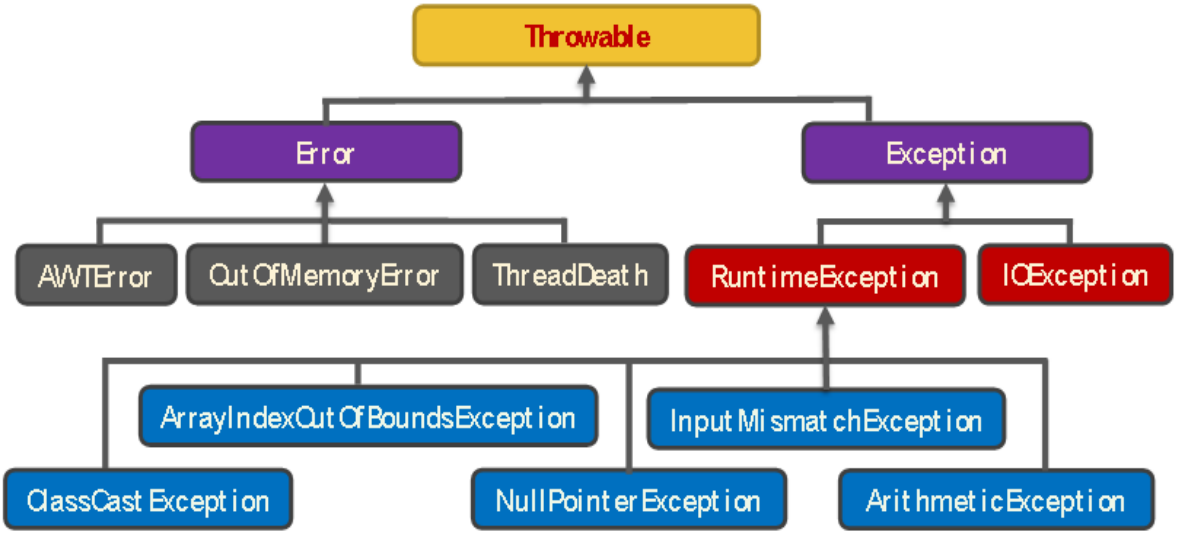


Figure 1.1: Hierarchy of Java Exception classes

a) **Errors**: Errors represent irrecoverable conditions such as JVM running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. These are not exceptions at all, but problems that arise beyond the control of the user or the programmer and we should not try to handle them.

b) **Exception**: Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception such as the name and description of the exception and the state of the program when the exception occurred.

Types of Java Exceptions

I. Checked Exception

A checked Exception is an exception that is checked by the compiler at the compile-time. Such exceptions are known as compile-time exceptions. These exceptions cannot simply be ignored, and the programmer is prompted to handle these exceptions.

Example, IOException, FileNotFoundException, etc.

II. Unchecked Exception

Unchecked exceptions are the class that extends RuntimeException class. They occur at the time of program execution. These are also known as Runtime Exceptions, and, they are ignored at the time of compilation.

Examples; ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. You can think about it this way. “If it is a runtime exception, it is your fault”.

What is"Java Exception Handling"?

The problem with the exception is, it terminates the program and skips the rest of the execution, i.e if a program has 10 lines of code and at line 6 an exception occurs then the program will terminate immediately by skipping the execution of the rest of 4 lines of code. To handle a such problem, we use exception handling that avoids program termination and continues the execution by skipping the exception code.

Java Exception Handling is a mechanism to handle runtime errors such as IOException, ClassNotFoundException, SQLException, etc. It provides a meaningful message to the user about the issue rather than a system-generated message.

Ways to handle exceptions in Java.

1. **try...catch block**

Syntax:

```
try {
        // suspected code
} catch (ExceptionClass ec) {}
```

Try and catch both are Java keywords and used for exception handling. The try block is used to enclose a suspected code (code that may raise an exception during program execution).

For example, if a code raises an arithmetic exception due to dividing by zero then wrap that code into the try block.

```
try{
    int a = 10, b = 0;
    int c = a/b;    // exception
}catch(ArithmeticException e){
    System.out.println(e);
}
```

The catch block handles the exception thrown by the code enclosed in the try block. The catch block must be used after the try block only.

2. **finally block**

The finally block is optional. And, for each try block, there can be only one finally block. A finally block of code is always executed whether an exception has occurred or not. It appears at the end of the catch block.

```
try {
     // code
} catch (ExceptionType1 e1) {
```

```
        // code
    } finally {}
```

3. throw and throws keyword

a) <u>throw keyword</u>

Used to throw a single exception explicitly. The only object of the Throwable class or its sub-classes can be thrown. When we throw an exception, the flow of the program moves from the try block to the catch block. Program execution stops on encountering a throw statement, and the closest catch statement is checked for a matching type of exception.

Syntax: throw ThrowableInstance

We allow using a **new** operator to create an instance of the class Throwable

```
new ArithmeticException (“divideByZero");
```

This constructs an instance of ArithmeticException with the name divideByZero. In the following example, the divideByZero () method throw an instance of ArithmeticException, which is successfully handled using the catch statement, and thus, the program prints the output "Trying to divide by 0".

Example: **Exception handling using Java throw**

```
class Main {
    public static void divideByZero() {
        try {
            throw new ArithmeticException("divideByZero");
        } catch(ArithmeticException e) {
            System.out.println("Trying to divide by 0”);
        }
    }
    public static void main(String[] args) {
        divideByZero();
    }
}
```

<u>OUTPUT</u>

Trying to divide by 0

b) <u>throws keyword</u>

throws keyword used to declare the list of exceptions that a method may throw during the execution of the program. Any method that is capable of causing exceptions must list all the exceptions possible during its execution so that anyone calling that method gets prior knowledge about which exceptions are to be handled.

Syntax: type method_name (parameters) throws exception_list {
// definition of the method
}

// Declaring the type of exception

```
public static void findFile() throws IOException {
        // code that may generate IOException
        File newFile = new File("test.txt");
        FileInputStream stream = new FileInputStream(newFile);
}
public static void main(String[] args) {
        try{
            findFile();
        }catch (IOException e) {
            System.out.println(e);
        }
}
```

Chapter Two: Multithreading in Java

2.1 Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU. Java Multithreading is mostly used in games, animation, etc. A multithreading program contains two or more parts of a program that can run concurrently. Each part of a program is called a **thread**. So, a **thread** is a lightweight smallest part of a process that can run concurrently with other threads of the same process.

Threads are independent because they all have a separate path of execution that's the reason if an exception occurs in one thread, it doesn't affect the execution of other threads. Every Java program has at least one thread – called the **main** thread. All threads of a process share a common memory. JVM manages those threads and schedules them for execution.

Reasons for using Multithreading in Java

1. **Parallel Programming**

The main reason is to make a task run parallel to another task e.g. drawing and event handling. GUI applications e.g. Swing and Java FX GUIs are the best examples of multithreading in Java. In a typical GUI application, the user initiates an action e.g. downloading a file from the network or loading games modules from a hard disk.

2. **To take full advantage of CPU power**

The main reason is to improve the throughput of the application by utilizing full CPU power. **For example**, if you have got 32 core CPUs and you are only using 1 of them for serving 1000 clients and assuming that your application is CPU bound, you can improve throughput to 32 times by using 32 threads, which will utilize all 32 cores of your CPU.

3. **For reducing response time**

We can also use multiple threads to reduce response time by doing fast computation by dividing a big problem into smaller chunks and processing them by using multiple threads.

4. **To sever multiple clients at the same time**

Using multiple threads significantly improves an application's performance in a client-server application. A multi-threaded server means multiple clients can connect to the server at the same time. This means the next client doesn't have to wait until your application finish processing the request of the previous client. In the below example, multiple requests are processed by our multi-threaded server at the same time.

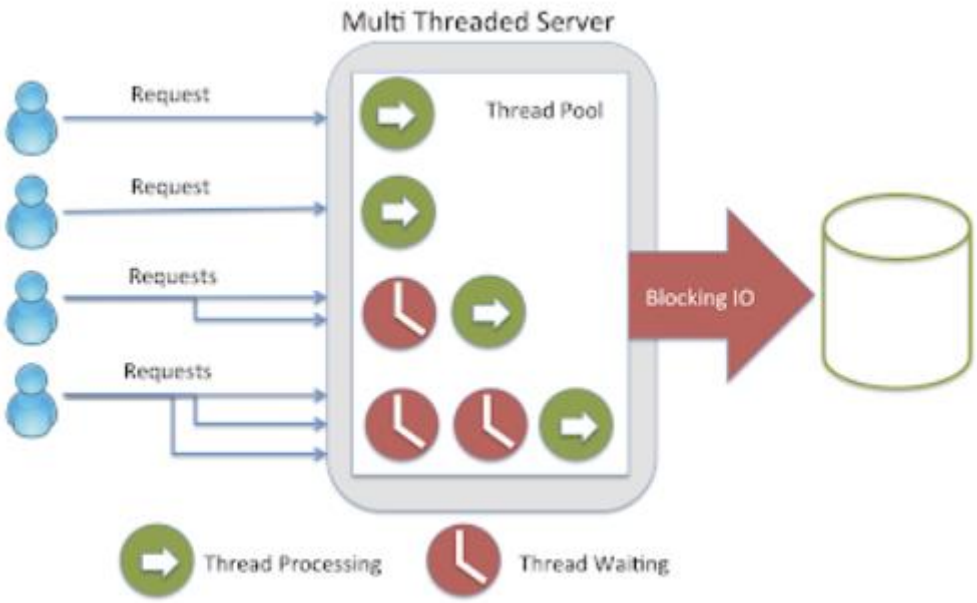


Figure 2.1 - Multithreaded Server

2.2 The life cycle of Thread

A thread moves several states in its life cycle. This life cycle is controlled by JVM. These possible states are:

i. New

In this phase, the thread is created using the class "Thread class". The thread is in the new state if you create an instance of the Thread class but before the invocation of the start() method. It is also known as the born thread. Transitions to Runnable state after it is started by start() method.

ii. Active

When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.

- **Runnable (ready):** In this stage, the instance of the thread is invoked with a start method.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state.

iii. Waiting (Non-Runnable/Blocked)

A thread enters this state when it is temporarily in inactive state. i.e., it is still alive but is not eligible to run. It can be in a waiting, sleeping, or blocked state.

iv. Dead (terminated)

A thread is terminated due to the following reasons: Either its run() method exists normally, i.e., the thread's code has executed the program. Or due to some unusual errors like segmentation fault or an unhandled exception.

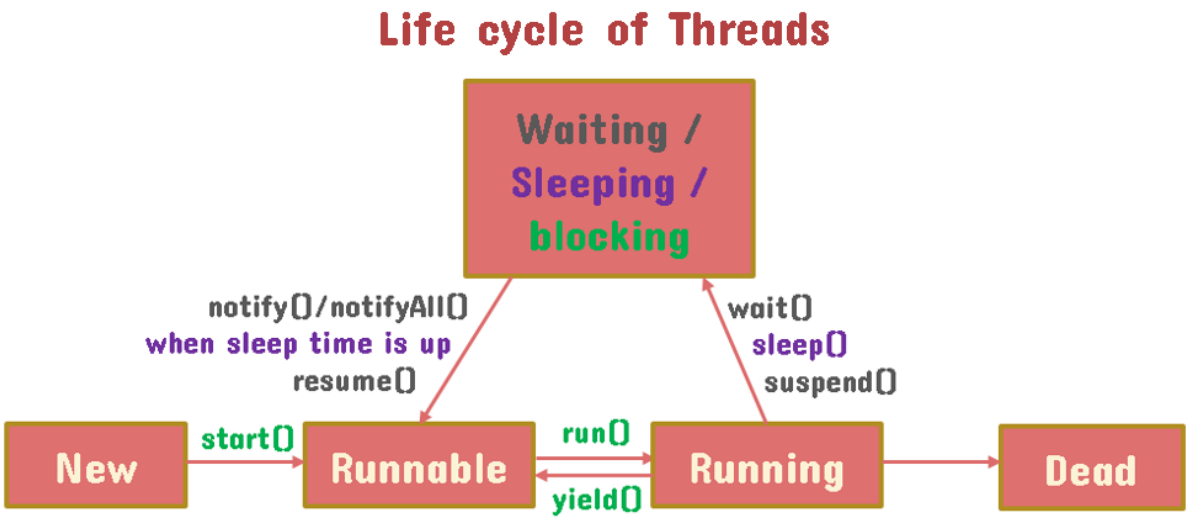


Figure 2.2 - Threads States

2.3 Thread Creation

There are two fundamental methods for creating a thread class:

I. Create a class that extends Thread

Create one class that extends the thread class. Override the run method and put lines of code inside the thread method that will be performed by a thread. Create an object of a class that we created by extending the Thread class and call the start() method to execute the thread.

```
class Multi extends Thread{
    public void run(){
        System.out.println ("thread is running...");
    }
    public static void main(String args[]){
        Multi thread1=new Multi();
        thread1.start();
    }
}
```

II. Create a class that implements a Runnable interface

```
class Multi3 implements Runnable{
    public void run(){
        System.out.println  ("thread is running...");
```

```
    }
    public static void main(String args[]){
      Multi3 m1=new Multi3();
      Thread thread1 =new Thread(m1);
      thread1.start();
    }
}
```

2.4 Thread Priority

Each java thread has its priority which decides the order of threads to be scheduled. The threads of equal priority will be given the same treatment by the java scheduler. And they will follow the FCFS (First Come First Serve) algorithm. Thread class defines three int constants that are used to specify the priority of a thread.

- MAX_PRIORITY (default is 10)
- MIN_PRIORITY (default is 1)
- NORM_PRIORITY (default is 5)

In any Thread the default priority is NORM_PRIORITY.

2.5 Daemon Threads

It is a low-priority thread that runs in the background to perform tasks such as garbage collection. Also, it is a service provider thread that provides services to the user thread.

2.6 Sharing Variables

For multiple threads to be useful in a program, they have to have some way to communicate or share their results. The simplest way for threads to share their results is to use shared variables.

Race condition & How to Solve it:

Race conditions occur when multiple, asynchronously executing threads access the same object (called a shared resource) returning unexpected (wrong) results. They can be avoided by synchronizing the threads which access the shared resource.

Synchronization: is needed in a multi-threading application to ensure that one thread is updating shared data and other threads wait and get the updated value. It uses a lock to represent the shared data to allow each thread to use the lock status to Synchronize with each other.

Synchronization is a programming technique that involves 3 elements:

- **Lock**: An object with two states: locked and unlocked.
- **Synchronized Block:** A block of statements that is associated with a lock.

- **Synchronization Rule**: When a synchronized block is encountered in a thread of execution, the associated lock will be checked.

Java also offers three ways to define synchronized blocks.

- **Method 1:** Static Synchronized Method:

```
class class_name {
        static synchronized return_type method_name() {
                //statement block
        }
}
```

- **Method 2:** Synchronized Method:

```
class class_name {
        synchronized return_type method_name() {
                //statement block
        }
}
```

- **Method 3:** Synchronized Block:

```
class class_name {
        return_type method_name() {
                synchronized (object) {
                        statement block
                }
        }
}
```

An important point of synchronized keywords in Java

1. **Synchronized** keyword in Java is used to provide mutually exclusive access to a shared resource with multiple threads in Java.
2. **Synchronization** in java guarantees that no two threads can execute a synchronized method that requires the same lock simultaneously or concurrently.
3. Whenever a thread enters into a java synchronized method or block it acquires a lock and whenever it leaves a java synchronized method or block it releases the lock.
4. One Major disadvantage of the java synchronized keyword is that it doesn't allow concurrent read which you can implement using java.util.concurrent.locks.ReentrantLock.
5. Java synchronized keyword incurs a performance cost.

6. You cannot apply java synchronized keywords with variables.

1. What is multithreaded programming?

A. It's a process in which two different processes run simultaneously

B. It's a process in which two or more parts of same process run simultaneously

C. It's a process in which many different process are able to access same information

D. It's a process in which a single process can access information from many sources

2. Thread priority in Java is?

A. Integer B. Float C. double D. long

3. What will happen if two thread of the same priority are called to be processed simultaneously?

A. Anyone will be executed first lexographically

B. Both of them will be executed simultaneously

C. None of them will be executed

D. It is dependent on the operating system

4. What is the valid range of priority of a thread in Java multi-threading?

A. 1 to 10 B. 0 to 10 C. 0 to 9 D. 1 to 9

5. We can create thread in java by

A. implementing Thread C. extending Runnable

B. extending Thread D. both b & c

6. What is the priority of the thread in the following Java Program?

```
class multithreaded_programing {
    public static void main(String args[]) {
            Thread t = Thread.currentThread();
            System.out.println(t);
    }
}
```

A. 0 B. 1 C. 10 D. 5

7. What is true about threading?

A. run() method calls start() method and runs the code

B. run() method creates new thread

C. run() method can be called directly without start() method being called

D. start() method creates new thread and calls code written in run() method

Chapter Three: Java GUI Development (SWING)

3.1Graphical User Interface (GUI)

GUI refers to an interface that allows one to interact with electronic devices like computers and tablets through graphic elements. GUI uses icons, menus and other graphical representations to display information, as opposed to text-based commands. GUIs are built from GUI components. These are sometimes called controls or widgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. It plays an important role to build easy interfaces for Java applications.

Java GUI APIs: Java has two GUI packages (Libraries).

1 Abstract Windows Toolkit (AWT)

2 Swing

A. Abstract Windows Toolkit (AWT)

When java was introduced, the GUI classes were bundled in a library known as Abstract Windows Toolkit (AWT). AWT is one the oldest GUI frameworks and is Java's original set of classes for building GUIs. It allows Java programmers to develop window-based applications.

Its components are platform-dependent i.e. components are displayed according to the view of operating system. It looks different & lays out inconsistently on different OS.

In simple words, an AWT application will look like a windows application in Windows OS whereas it will look like a Mac application in the MAC OS. It is heavy weight i.e. its components are using the resources of underlying operating system (OS).

The **java.awt** package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc. In addition, AWT is adequate for many applications but it is difficult to build an attractive GUI.

B. Swing

It is designed to solve AWT's problems (since Java 2). The components of the Swing library are easier to work with and are much better implemented. It is 99% java; It is lightweight components as drawing of components is done in java. It is a part of Java Foundation Classes (JFC) that is used to create window-based applications. JFC are a set of GUI components which simplify the development of desktop applications. Swing GUI components allow you

to specify a uniform look-and-feel for your application across all platforms. It also lays out consistently on all OSs. Some Swing components need classes from the AWT library.

It has much bigger set of built-in components and uses AWT event handling. Swing classes are located in Swing is built "on top of" AWT, so you need to import AWT and use a few things from it.

- Swing is bigger and slower.
- Swing is more flexible and better looking.
- Swing and AWT are incompatible. Thus, you can use either, but you can't mix them.
- package javax.swing.

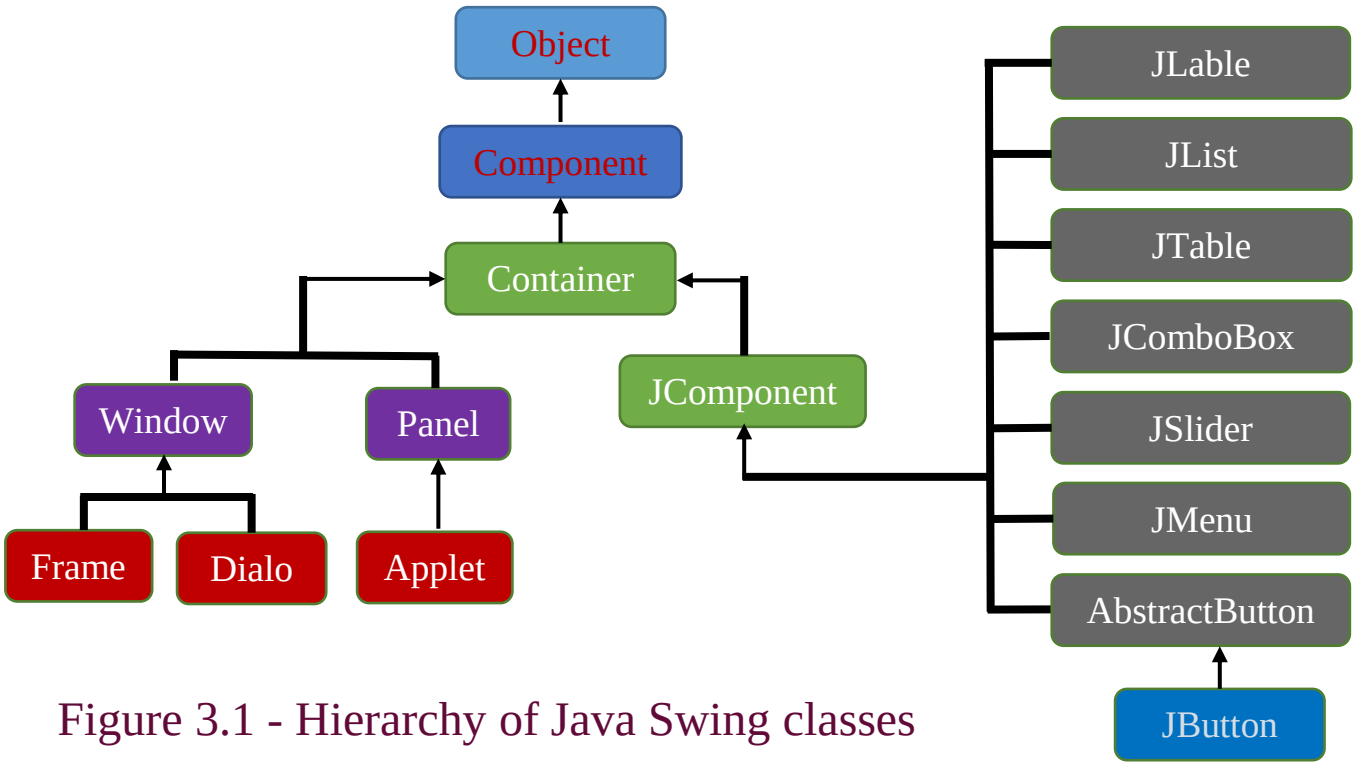


Figure 3.1 - Hierarchy of Java Swing classes

As seen from the above hierarchy we have Container classes – Frame, Dialog, Panel, Applet, etc. There are also Component classes derived from the JComponent class of Swing API. Some of the classes that inherit from JComponent are JLabel, JList, JTextBox, etc.

1. **Containers**

In Java, Containers are divided into two types

- ❖ **Top-level Container** (JFrame, JDialog, JApplet)
  - Inherited by Component and Container of AWT.
  - They are heavyweight & other containers cannot hold it.
- ❖ **Lightweight Container** (JPanel)

- They inherit JComponent class and used as general purpose container.
- Used to keep and organize related components together.

i. JFrame

- A JFrame is a subclass of Window. It has a border, menu bar, title bar.
- JFrame class has its default layout as BorderLayout.
- It can hold all other components like button, text field, checkboxes, scrollbar etc.
- JFrame is the most commonly used container while developing Swing application.

ii. JDialog

- JDialog is a subclass of Window. JDialog class has border and title.
- To create a dialog object, an instance of the associated JFrame class is always needed.

iii. JPanel

- The JPanel is the container that doesn't contain title bar, border or menu bar.
- It is generic container for holding the components.
- It can have other components like button, textfield etc.
- An instance of Panel class creates a container, in which we can add components.
- Panel has FlowLayout as its default layout.

The JFrame class has the following two constructors:

- **JFrame():** This constructor constructs a new frame with no title and it is initially invisible.
- **JFrame (String title):** This constructor constructs a new, initially invisible frame with specified title.

Some methods of JFrame class:

- **void add(JComponent comp):** to add several items to the frame.
- **void setSize**(**int width**, **int height**): to specify the size of the frame.
- **void setLocation(int x, int y):** to specify the upper left corner location of the frame.
- **void setDefaultCloseOperation(int mode):** set the operation that will happen by default when the user initiates a "close" on this frame. Possible values are:
    - JFrame.EXIT_ON_CLOSE: exit the application.
    - JFrame.HIDE_ON_CLOSE: Hide the frame, but keep the application running.
    - JFrame.DO_NOTHING_ON_CLOSE: Ignore the click.

- **void setVisible(boolean b)**: to show or hide the frame depending on the value of parameter b.
- **void setBounds(int x, int y, int width, int height):** to specifies the size of the frame and the location of the upper left corner.
- **void setTitle**(**String title**): to set the title for the frame by the specified string.

```
import javax.swing.JFrame;
public class Simple{
   public static void main(String[] args) {
            JFrame f=new JFrame("EmptyFrame");
            f.setSize(300,200);
            f.setTitle("First JFrame");
            f.setLocationRelativeTo(null);
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            f.setVisible(true);
   }
}
```

2. JComponent

The JComponent class is the base class of all Swing components except top-level containers. Swing components whose names begin with "J" are descendants of the JComponent class. For example, JButton, JScrollPane, JPanel, JTable etc. But, JFrame and JDialog don't inherit JComponent class because they are the child of top-level containers.

Here is a list of controls in the javax.swing package.

❖ **Input Components**

- Buttons (JButton, JRadioButtons, JCheckBox)
- Text (JTextField, JTextArea)
- Menus (JMenuBar, JMenu, JMenuItem)
- Sliders (JSlider)
- JComboBox (uneditable) (JComboBox)
- List (JList)

❖ **Choosers**

- File chooser (JFileChooser)

- Color chooser (JColorChooser)

❖ **Information Display Components**

- Label (JLabel)
- Progress bars (JProgressBar)
- Tool tips (using JComponent's setToolTipText(s) method)

❖ **More complex displays**

- Tables (JTable)
- Trees (JTree)

Java Event Handling

Event handling is a mechanism through which an action is taken when an event occurs. For this, we define a method which is also called an event handler that is called when an event occurs. Java uses a standard mechanism called the "**Delegation event model**" to generate as well as handle events.

The Delegation event model consists of…

❖ **Source**

- A **source** is an object that generates an event. It is responsible for sending information about the event to the event handler.

❖ **Listener**

- It is an object that is notified when an event occurs.
- It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events.
- Second, it must implement methods to receive and process these notifications.
- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**.

The **java.awt.event** package provides many event classes and Listener interfaces for event handling.

Some Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener

TextEvent	TextListener

The following two steps are required to perform event handling:

a) Register a listener object with a source object

```
eventSourceListener.addEventListener(eventListenerObject);
```

b) Create the listener object. Create a class that implements that interface

```
class MyListener implements ActionListener{
    public void actionPerformed(ActionEvent e){
        //code to run
    }
}
```

3. JLabel

A **label** is a display area for a short text (a non-editable), an image, or both.

A JLabel component can be created using one of the following constructors:

- **JLabel**(): Creates a default label with no text and icon.
- **JLabel**(String text): Creates a label with text.
- **JLabel**(Icon icon): Creates a label with an icon.
- **JLabel**(String text, int horizontalAlignment): Creates a label with a text and the specified horizontal alignment.
- **JLabel**(Icon icon, int horizontalAlignment): Creates a label with an icon and the specified horizontal alignment.
- **JLabel**(String text, Icon icon, int horizontalAlignment): Creates a label with text, an icon, and the specified horizontal alignment.

```
import javax.swing.*;
class LabelExample{
    public static void main(String args[]){
        JFrame f= new JFrame("Label Example");
        JLabel l1=new JLabel("First Label.");
        l1.setBounds(50,50, 100,30);
        f.add(l1);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

4. JButton

A **button** is a component that triggers an action event when clicked.

A Jbutton component can be created using one of the following constructors.

- **JButton():** Creates a default button with no text and icon.
- **JButton(Icon icon):** Creates a button with an icon.
- **JButton(String text):** Creates a button with text.
- **JButton(String text, Icon icon):** Creates a button with text and an icon.

```
import javax.swing.*;
        public class ButtonExample {
                public static void main(String[] args) {
                JFrame f=new JFrame("Button Example");
                JButton b=new JButton("Click Here");
                b.setBounds(50,100,95,30);
                f.add(b);
                f.setSize(400,400);
                f.setLayout(null);
                f.setVisible(true);
        }
}
```

5. JTextField

A text field is a box that contains a line of text. The user can type text into the box and the program can get it and then use it as data. The program can write the results of a calculation to a text field. Text fields are useful in that they enable the user to enter in variable data (such as a name or a description). **JTextField** is swing class for an editable text display.

A JTextField component can be created using one of the following constructors:

- **JTextField():** it creates a default empty text field with number of columns set to 0.
- **JTextField(int columns):** it creates an empty text field with the specified number of columns.
- **JTextField(String text):** it creates a text field initialized with the specified text.
- **JTextField(String text, int columns):** it creates a text field initialized with the specified text and the column size.

```
import javax.swing.*;
class TextFieldExample{
        public static void main(String args[]){
                JFrame f= new JFrame("TextField Example");
                JTextField t1,t2;
                t1=new JTextField("Welcome to IT Class.");
                t1.setBounds(50,100, 200,30);
                t2=new JTextField("AWT Tutorial");
                t2.setBounds(50,150, 200,30);
                f.add(t1);
                f.add(t2);
                f.setSize(400,400);
                f.setLayout(null);
                f.setVisible(true);
        }
}
```

6. JTextArea

JTextArea class, enables the user to enter multiple lines of text.

JTextArea components can be created using one of the following constructors:

- **JTextArea(int rows, int columns):** creates a text area with the specified number of rows and columns.
- **JTextArea(String s, int rows, int columns):** creates a text area with the initial text and the number of rows and columns specified.

```
import javax.swing.*;
public class TextAreaExample{
        TextAreaExample(){
                JFrame f= new JFrame();
                JTextArea a=new JTextArea("Welcome to IT Class");
                a.setBounds(10,30, 200,200);
                f.add(a);
                f.setSize(300,300);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main(String args[]){
                new TextAreaExample();
        }
}
```

3.2 Layout Management

Layouts tell Java where to put components in containers (JPanel, content pane, etc.). Layout manager is created using LayoutManager class. Every layout manager class implements the LayoutManager class. Each layout manager has a different style of positioning components. If you don't specify otherwise, the container will use a default layout manager. Every panel (and other container) has a default layout, but it's better to set the layout explicitly for clarity. Layout managers are set in containers using the setLayout(LayoutManager) method.

- **Example**: assume we have a Jframe instance called container.
- If we are going to add controls inside this container and if we are interested on the flowlayout, we can set as follow.

LayoutManager layoutManager = new FlowLayout();

container.setLayout(layoutManager);

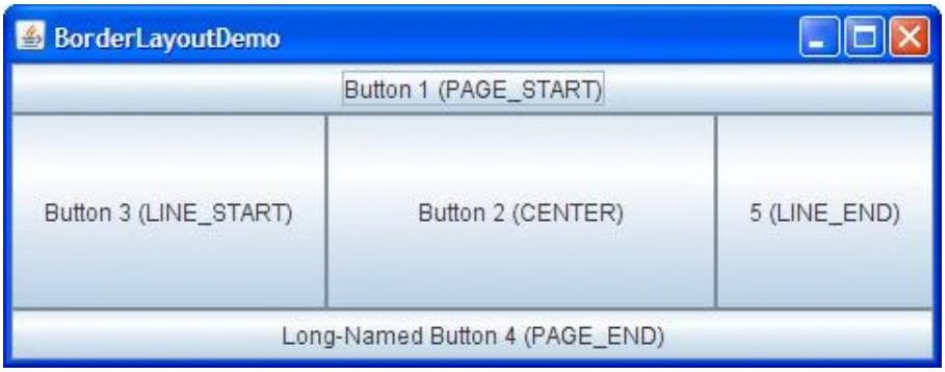
OR

container.setLayout(new FlowLayout());

Java supplies several layout managers including

A. Java BorderLayout

A BorderLayout places components in up to five areas: East, South, West, North, and Center. It is the default layout manager for every java JFrame



Components are added to a BorderLayout by using add(Component, index) where index is a constant such as:

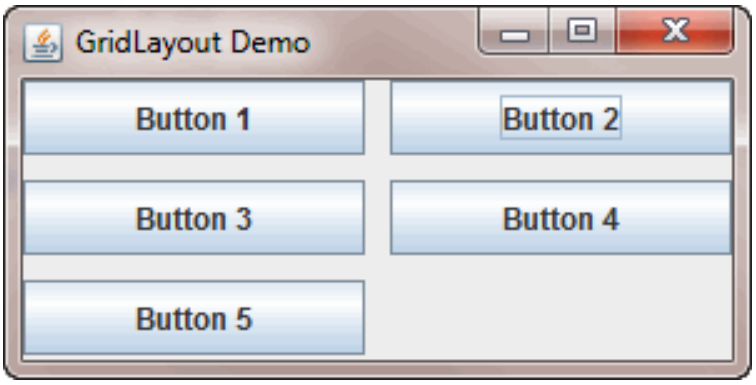
- BorderLayout.EAST
- BorderLayout.SOUTH
- BorderLayout.WEST
- BorderLayout.NORTH
- BorderLayout.CENTER

B. Java FlowLayout

FlowLayout is the Simplest and the default layout manager for every JPanel. It simply lays out components in a single row one after the other.

C. Java GridLayout

The GridLayout manager divides the container up into a given number of rows and columns. All sections of the grid are equally sized and as large as possible.



1. Give the abbreviation of AWT?

   A. Applet Windowing Toolkit
   B. Abstract Windowing Toolkit
   C. Absolute Windowing Toolkit
   D. None of the above

2. Where are the following four methods commonly used?

   i. public void add(Component c)
   ii. public void setSize(int width,int height)
   iii. public void setLayout(LayoutManager m)
   iv. public void setVisible(boolean)

   A. Graphics class B. Component class C. Both A & B D. None

3. Implement the Listener interface and overrides its methods is required to perform in event handling.

   A. True B. False

4. Which is the container that doesn't contain title bar and MenuBars but it can have other components like button, textfield etc.

   A. Window B. Frame C. Panel D. Container

5. The Java Foundation Classes (JFC) is a set of GUI components which simplify the development of desktop applications.

   A. True B. False

6. Which are passive controls that do not support any interaction with the user?

   A. Choice B. List C. Labels D. Checkbox

7. Which package provides many event classes and Listener interfaces for event handling?

   A. java.awt B. java.awt.Graphics C. java.awt.event D. None

Chapter Four: Java Networking

4.1 Java Networking

Java Networking is a concept of connecting two or more computing devices together to share the resources. Java socket programming provides facility to share data between different computing devices. Java Program communicates over the network at the application layer.

java.net package is useful for all the Java networking classes and interfaces. java.net package encapsulate large number of classes and interface that provides an easy-to use means to access network resources.

The java.net package supports two protocols,

- **Transmission Control Protocol(TCP)**
  - It provides reliable communication between the sender and receiver.
  - TCP is used along with the Internet Protocol referred as TCP/IP.

- **User Datagram Protocol(UDP)**
  - It provides a connection-less protocol service by allowing packet of data to be transferred along two or more nodes.

Advantage of Java Networking

- Sharing resources
- Centralize software management

4.2 Networking Terminologies in Java

1) **IP Address**
   - The IP address is a unique number assigned to a node of a network e.g. 192.168.0.1.
   - It is composed of octets that range from 0 to 255.
   - It is a logical address that can be changed.

2) **Protocol**
   - A protocol is an organized set of communication rules that determine how data is transmitted between devices on the same network.
   - For example: TCP, FTP, Telnet, SMTP, POP etc.

3) **Port Number**
   - The port number uniquely identifies a particular process or service on a system.

- It acts as a communication endpoint between applications.
- To communicate between two applications, the port number is used along with an IP Address.

4) MAC (Media Access Control) Address

- It is basically a hardware identification number which uniquely identifies each device on a network.
- A network node can have multiple Network Interface Controller(NIC) but each with unique MAC address.
- For example, an Ethernet card may have a MAC address of 00:0d:83::b1:c0:8e.

5) Connection-oriented and connection-less protocol

In connection-oriented protocol

- Acknowledgement is sent by the receiver.
- It is reliable but slow.
- Example of connection-oriented protocol is TCP.

In connection-less protocol

- Acknowledgement is not sent by the receiver.
- It is not reliable but fast.
- Example of connection-less protocol is UDP.

6) Socket

- A socket is an endpoint between two way communications and it used to identify both a machine and a service within the machine.
- It is tied to a port number so that TCP layer can recognize the application to which data is intended to be sent.

4.3 Java Socket Programming

It is used for communication between the applications running on different JRE. It can be connection-oriented or connection-less.

In Socket Programming, Socket and ServerSocket classes are used for connection-oriented, and DatagramSocket and DatagramPacket classes are used for connection-less socket programming. ServerSocket class is for servers and Socket class is for the client.

The client in socket programming must know two information: IP Address of Server, and Port number.

1. Which of the following protocol follows connection less service?

   A. TCP B. TCP/IP C. UDP D. HTTP

2. Which of the following statement is NOT true?

   A. TCP is a reliable but slow.

   B. UDP is not reliable but fast.

   C. File Transfer Protocol (FTP) is a standard Internet protocol for transmitting files between computers on the Internet over TCP/IP connections.

   D. In HTTP, all communication between two computers are encrypted

3. Which of the following statement is **TRUE**?

   A. With stream sockets there is no need to establish any connection and data flows between the processes are as continuous streams.

   B. Stream sockets are said to provide a connection-less service and UDP protocol is used

   C. Datagram sockets are said to provide a connection-oriented service and TCP protocol is used.

   D. With datagram sockets there is no need to establish any connection and data flows between the processes are as packets.

4. Which class is used to create servers that listen for either local client or remote client programs?

   A. ServerSocketsB. httpServer C. httpResponse D. None

5. Which classes are used for connection-less socket programming?

   A. DatagramSocket B. DatagramPacket C. Both A & B D. None

Chapter Five: Java Database Connectivity (JDBC)

5.1 Java Database Connectivity (JDBC)

In most Java applications, there is always a need to interact with databases to retrieve, manipulate, and process the data. For this purpose, Java JDBC has been introduced. So that, JDBC is an API (Application programming interface) used for interacting with databases in Java programming. By using JDBC, we can interact with different types of Relational Databases such as Oracle, MySQL, MS Access, Sybase etc.

Before JDBC, ODBC API was introduced to connect and perform operations with the database. ODBC uses an ODBC driver which is platform-dependent because it was written in the C programming language. But, JDBC API that written in Java language, is platform-independent, and makes Java platform-independent itself.

JDBC API uses JDBC drivers to connect with the database, and it acts as an interface between the Java program and Database.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. The **java.sql** package contains classes and interfaces for JDBC API.

5.2 JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. It convert requests from Java programs to a protocol that the DBMS can understand.

There are 4 types of JDBC Drivers available. They are

- **Type I** : **JDBC-ODBC Bridge Driver**
- **Type II**: **Native API Driver** *(Partially Java Driver)*
- **Type III**: **Network Protocol Driver** *(Fully Java Driver)*
- **Type IV**: **Thin Driver** *(Fully Java Driver)*

Type I: <u>JDBC-ODBC Bridge Driver</u>

- In this type of Driver, **JDBC–ODBC** Bridge act as an interface between client and database (DB) server.

- When a user uses Java application to send requests to the database using JDBC–ODBC Bridge, it first converts the JDBC API to ODBC API and then sends it to the database.
- When the result is received from DB, it is sent to ODBC API and then to JDBC API.
- This Driver is platform-dependent because it uses ODBC which depends on the native library of the OS.
- In this Type, JDBC–ODBC driver should be installed in each client system & the DB must support ODBC driver.

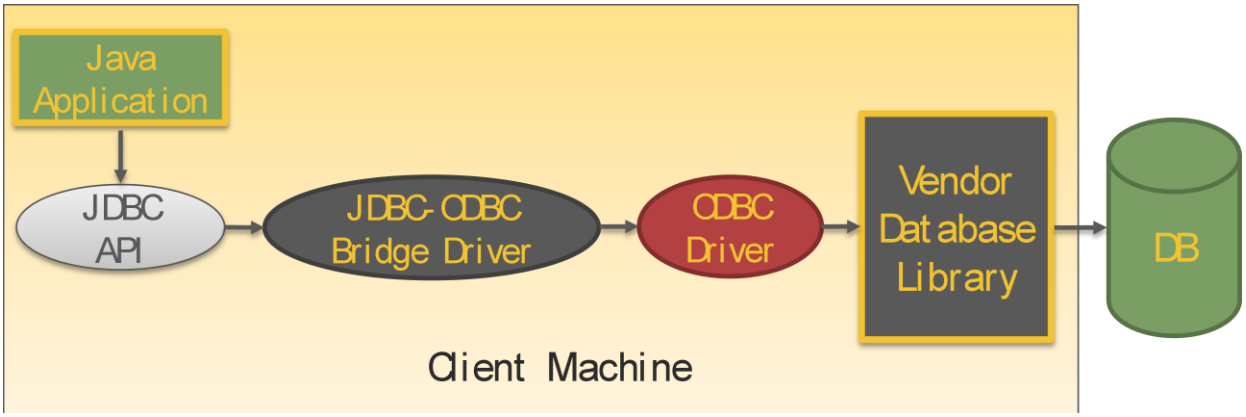


Figure 5.1 - JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Type II: <u>Native API Driver</u>

- It is similar to Type I Driver. Here, the ODBC part is replaced with native code in Type II Driver. This native code part is targeted at a specific database product.
- It uses the libraries of the client-side of the database.
- This Driver converts the JDBC method calls to native C/C++ method calls of the database native API.

- When the database gets the requests from the user, the requests are processed and sent back with the results in the native format which are to be converted to JDBC format and give it to the Java application.
- This type of driver gives faster response and performance than the Type I driver.

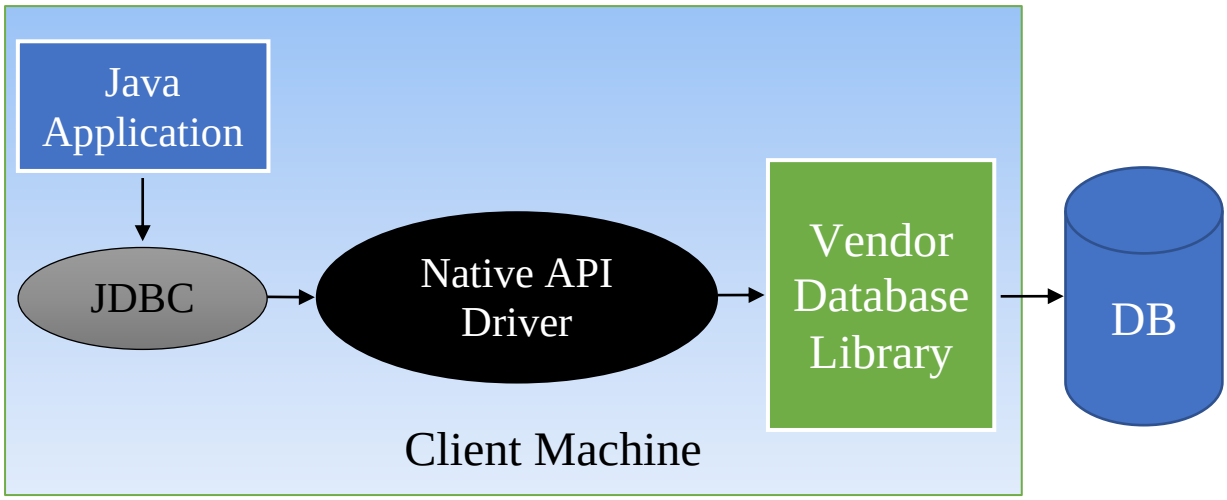


Figure 5.2 - Native API Driver

Advantage:

- Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Type III: <u>Network Protocol Driver</u>

- The type III driver is fully written in Java. It is like a 3-tier approach to access the database.
- It sends the JDBC method calls to an intermediate server. On behalf of the JDBC, the intermediate server communicates with the database.
- The Application server (intermediate or middle – tier) converts the JDBC calls either directly or indirectly to the vendor-specific Database protocol.

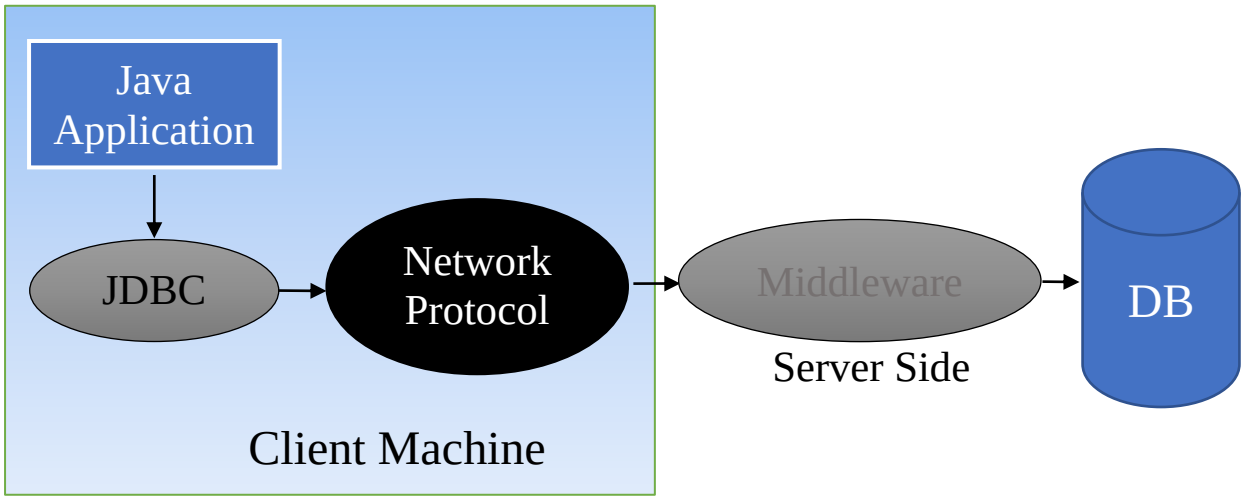


Figure 5.3 - Network Protocol Driver

Advantage:

- No client side library is required because of application server.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Type IV: Thin Driver

- **Thin driver** is directly implemented that converts JDBC calls directly into vendor-specific Database protocol.
- Today, most of the JDBC Drivers are type IV drivers. It is written fully in Java and thus it is platform-independent.
- It is installed inside the JVM (Java Virtual Machine) of the client, so you don't have to install any software on the client or server-side.
- This driver architecture has all the logic to communicate directly with the DB in a single driver.
- It provides better performance than the other type of drivers. It allows for easy deployment.

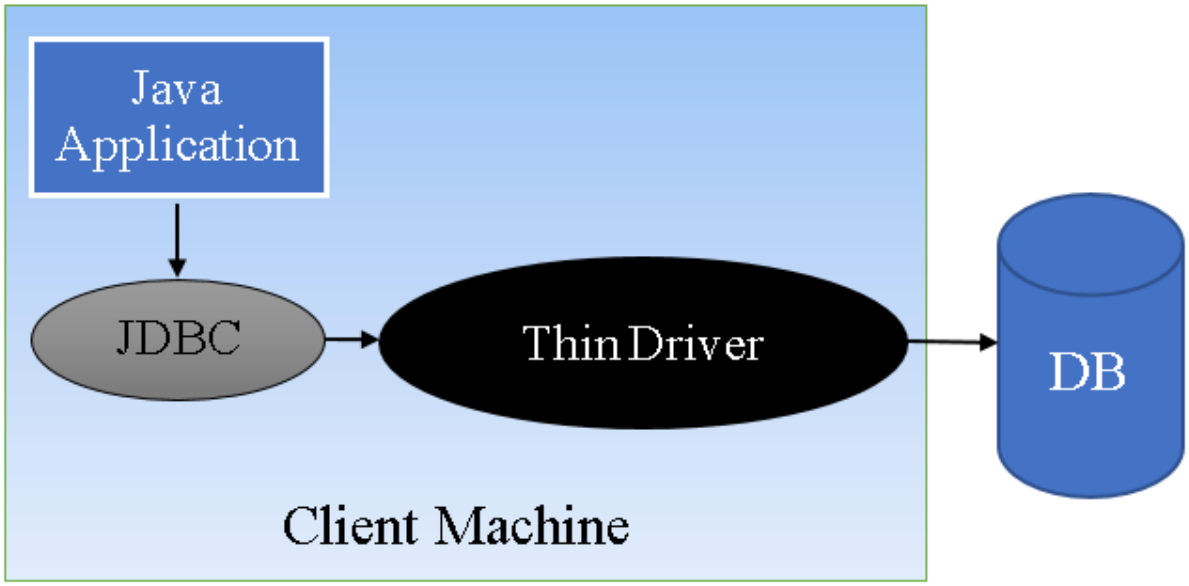


Figure 5.4 - Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

5.3 JDBC Connection Steps

There are 6 basic steps to connect with JDBC. They are

Step 1 – Import Packages

- Irrespective of the JDBC Driver, add the following import statement in the Java program. import java.sql.*;
- This package provides classes and interfaces to perform most of the JDBC functions like creating and executing SQL queries.

Step 2 - Load Driver

- Load/register the driver in the program before connecting to the Database.
- You need to register it only once per database in the program.

We can load the driver in the following 2 ways

i. Class.forName()

The forName() method of Class class is used to register the driver class. In this way, the driver's class file loads into the memory at runtime. It explicitly loads the driver.

ii. DriverManager.registerDriver()

DriverManager is an inbuilt class that is available in the java.sql package. Before you connect with the database, you need to register the driver with DriverManager. The main function of DriverManager is to load the driver class of the Database and create a connection with DB. It implicitly loads the driver. While loading, the driver will register with JDBC automatically. **Example**; DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())

Step 3 - Establish Connection

- DriverManager class has the getConnection method, that used to establish connection with the database.
- To call getConnection() method, we need to pass 3 parameters, such as URL, a username, and a password with string data type to access the database.
- The getConnection() method is an overloaded method.

The 2 methods are: getConnection(URL, username, password); **and** getConnection(URL);

Step 4 - Create Statement and Execute SQL queries

- Once the connection has established, we can interact with the connected Database. First, we need to create the statement to perform the SQL query and then execute the statement.

ii. Create Statement

In order to send the SQL commands to database from our java program, we need Statement object. We can get the Statement object by calling the createStatement() method on connection.

There are 3 Statement interfaces are available in the java.sql package.

a. Statement

- This interface is used to implement simple SQL statements with no parameter.
- It returns the ResultSet object.

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

b. PreparedStatement

- This interface extends the Statement interface. So, it has more features than the Statement interface.
- It is used to implement parameterized and precompiled SQL statements.
- **Example** – String s_query = “Select * from states where sid = 1”;
  PreparedStatement pst = conn.prepareStatement(s_query);

c. CallableStatement

- This interface extends the PreparedStatement interface.
- So, it has more features than the PreparedStatement interface.
- It is used to implement a parameterized SQL statement that invokes procedure or function in the database.
- A stored procedure works like a method or function in a class.
- It supports the IN and OUT parameters.
- The CallableStatement instance is created by calling the prepareCall method of the Connection object.
- **Example -** CallableStatement cs = con.prepareCall("{call procedures(?,?)}");

iii. Execute the SQL Query

There are 4 important methods to execute the query in Statement interface.

a. ResultSet executeQuery(String sql)

- The executeQuery() method in Statement interface is used to execute the SQL query and retrieve the values from DB.
- It returns the ResultSet object, that can be used to get all the records of a table.
- Normally, we will use this method for the SELECT query.

b. executeUpdate(String sql)

- The **executeUpdate()** method is used to execute value specified queries like INSERT, UPDATE, DELETE (DML statements), or DDL statements that return nothing.
- Mostly, we will use this method for inserting and updating.

c. execute(String sql)

- The **execute()** method is used to execute the SQL query.
- It returns true if it executes the SELECT query. And, it returns false if it executes INSERT or UPDATE query.

d. executeBatch()

- This method is used to execute a batch of SQL queries to the Database and if all the queries get executed successfully, it returns an array of update counts.
- We will use this method to insert/update the bulk of records.

Step 5 - <u>Retrieve Results</u>

- When we execute the queries using the **executeQuery()** method, the result will be stored in the ResultSet object.
- The returned ResultSet object will never be null even if there is no matching record in the table.
- ResultSet object is used to access the data retrieved from the Database.

```
ResultSet rs = st.executeQuery(QUERY);
```

- We can use the executeQuery() method for the SELECT query.
- When someone tries to execute the insert/update query, it will throw SQLExecption with the message “executeQuery method can not be used for update”.
- A ResultSet object points to the current row in the Resultset.
- To iterate the data in the ResultSet object, call the **next()** method in a while loop. If there is no more record to read, it will return FALSE.
- ResultSet can also be used to update data in DB.

- We can get the data from ResultSet using getter methods such as getInt(), getString(), getDate(). We need to pass the column index or column name as the parameter to get the values using Getter methods.

Step 6 - <u>Close Connection</u>

- Finally, we need to make sure that we have closed the resource after we have used it.
- If we don't close them properly we may end up out of connections.
- When we close the Connection object, Statement and ResultSet objects will be closed automatically.

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

1. What is JDBC?
   A. JDBC is a java based protocol.
   B. JDBC is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.
   C. JDBC is a specification to tell how to connect to a database.
   D. Joint Driver for Basic Connection

2. Which driver uses ODBC driver to connect to the database?
   A. JDBC-ODBC bridge driver
   B. Native - API driver
   C. Network Protocol driver
   D. Thin driver

3. Which of the following type of JDBC driver, is also called Type 2 JDBC driver?
   A. JDBC-ODBC Bridge plus ODBC driver
   B. Native-API, partly Java driver
   C. JDBC-Net, pure Java driver
   D. Native-protocol, pure Java driver

4. Which of the following holds data retrieved from a database after you execute an SQL query using Statement objects.
   A. ResultSet B. JDBC driver C. Connection D. Statement

5. Which of the following type of JDBC driver, uses database native protocol?
   A. JDBC-ODBC Bridge plus ODBC driver
   B. Native-API, partly Java driver
   C. JDBC-Net, pure Java driver
   D. Native-protocol, pure Java driver

6. Which driver converts JDBC calls directly into the vendor-specific database protocol?
   A. Native - API driver
   B. Network Protocol driver
   C. Thin driver
   D. Both B & C

7. Which of the following step establishes a connection with a database?
   A. Import packages containing the JDBC classes needed for database programming.
   B. Register the JDBC driver, so that you can open a communications channel with the database
   C. Open a connection using the DriverManager.getConnection () method.
   D. Execute a query using an object of type Statement.