

## APPLET FUNDAMENTALS

**Applets** are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

Let's begin with the simple applet shown here:

### *Example 1: Simple Applet*

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

This applet begins with two **import** statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical user interface. The second **import** statement imports the **applet** package, which contains the class **Applet**. Every applet that you create must be a subclass of **Applet**. The next line in the program declares the class **SimpleApplet**. This class must be declared as **public**, because it will be accessed by code that is outside the program. Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet. **paint( )** is called each time that the applet must redisplay its output. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

The applet does not have a **main( )** method. Unlike Java programs, applets do not begin execution at **main( )**. After you enter the source code, compile the applet in the same way that you have been compiling programs. Running an applet involves a different process.

In fact, there are two ways in which you can run an applet:

- Executing the applet within a Java-compatible web browser.
- Using an applet viewer, such as the standard tool, **appletviewer**.

An applet viewer executes applet in a window. This is generally the fastest and easiest way to test an applet. To execute an applet in a web browser, you need to write a short HTML text file that contains a tag that loads the applet. Currently, SunMicrosystems recommends using the APPLET tag for this purpose.

Here is the HTML file that executes **SimpleApplet**:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

The **width** and **height** statements specify the dimensions of the display area used by the applet. After you create this file, you can execute your browser and then load this file, which causes **SimpleApplet** to be executed.

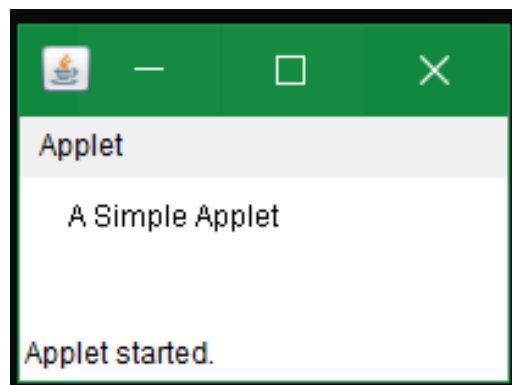
More convenient method for applets is simply including a comment at the head of the java source code file that contains an <applet> tag.

Therefore the **SimpleApplet** code looks like this:

**Example 2: Simple Applet**

```
import java.awt.*;
import java.applet.*;
/*<applet code="SimpleApplet" width=200 height=60>
</applet> */
public class SimpleApplet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

**Output**



## THE APPLET CLASS

The **Applet** class defines the methods that provide all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. **Applet** extends the AWT class **Panel**. In turn, **Panel** extends **Container**, which extends **Component**. These classes provide support for Java's window-based, graphical interface. Thus, **Applet** provides all of the necessary support for window-based activities.

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, **init()**, **start()**, **stop()**, and **destroy()**, apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. AWT-based applets (such as those discussed in this chapter) will also override the **paint()** method, which is defined by the AWT **Component** class. This method is called when the applet's output must be redisplayed.

These five methods can be assembled into the skeleton shown here:

### *Example 3: Applet Skelton*

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*<applet code="AppletSkel" width=300 height=100>
</applet>*/
public class AppletSkel extends Applet
{
    // Called first.
    public void init()
    {
        // initialization
    }
    /* Called second, after init().
    Also called whenever the applet is restarted. */
    public void start()
    {
        // start or resume execution
    }
    // Called when the applet is stopped.
```

```

    public void stop()
    {
        // suspends execution
    }

    /* Called when applet is terminated. This is the last method executed. */
    public void destroy()
    {
        // perform shutdown activities
    }

    // Called when an applet's window must be restored.
    public void paint(Graphics g)
    {
        // redisplay contents of window
    }
}

```

## **APPLET INITIALIZATION AND TERMINATION**

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

1. **init()**
2. **start()**
3. **paint()**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop()**
2. **destroy()**

Let's look more closely at these methods.

### **init()**

The **init()** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

### **start()**

The **start()** method is called after **init()**. It is also called to restart an applet after it has been stopped. Whereas **init()** is called once—the first time an applet is loaded—**start()** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start()**.

**paint()**

The **paint()** method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint()** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint()** is called. The **paint()** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

**stop()**

The **stop()** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop()** is called, the applet is probably running. You should use **stop()** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start()** is called if the user returns to the page.

**destroy()**

The **destroy()** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop()** method is always called before **destroy()**.

**APPLET LIFE CYCLE**

Every Java applet inherits a set of default behaviours from the **Applet class**. As a result when an applet is loaded it undergoes a series of changes in its state as shown in the figure 1.

The applet states include

- Born or initialization state
- Running state
- Idle state
- Dead or destroyed state

**Initialization state**

Applet enters initialization state when it is first loaded. This is achieved by calling the **init()** method of **Applet** class. The applet is born. At this stage initial values of applet is set by overriding **init()** method. The initialization occurs only once in the applet's life cycle.

**Idle or stopped state**

An applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also do so by calling the **stop()** method explicitly.

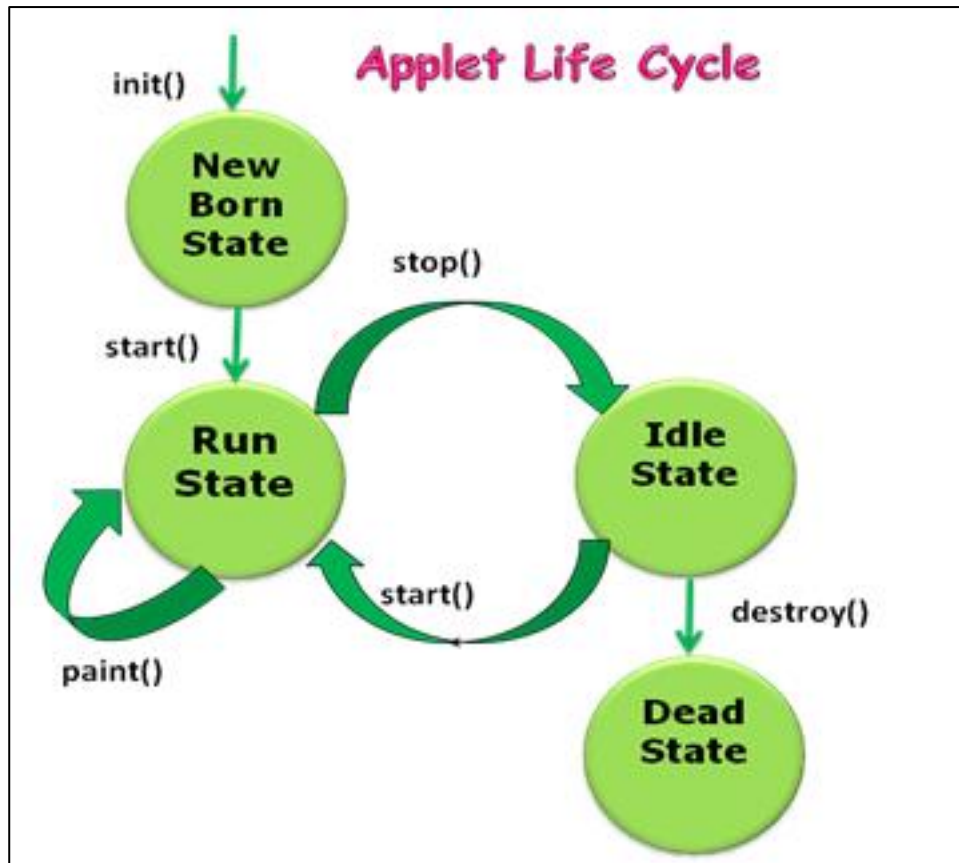


Figure 1 : Applet Life cycle

**Dead State**

An applet is said to be dead when it is removed from memory. This occurs automatically by invoking **destroy()** when we quit the browser. Like initialization, destroy stage occurs only once in the applet's life cycle.

**Display state**

Applet moves to the display state whenever it has to perform some output operations on the screen. This happens immediately after the applet enters into running state. The paint method is called to accomplish this task. Almost every applet will have a **paint()** method. Default version of paint() method does absolutely nothing. Therefore programmer need to override this method if anything to be displayed on screen.

**PASSING PARAMETERS TO APPLETS**

User defined parameters can be supplied to an applet using `<PARAM>` tags. The PARAM tag allows you to specify applet-specific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method. It returns the value of the specified parameter in the form of a **String** object.

***Example 4: Simple Applet retrieving parameter***

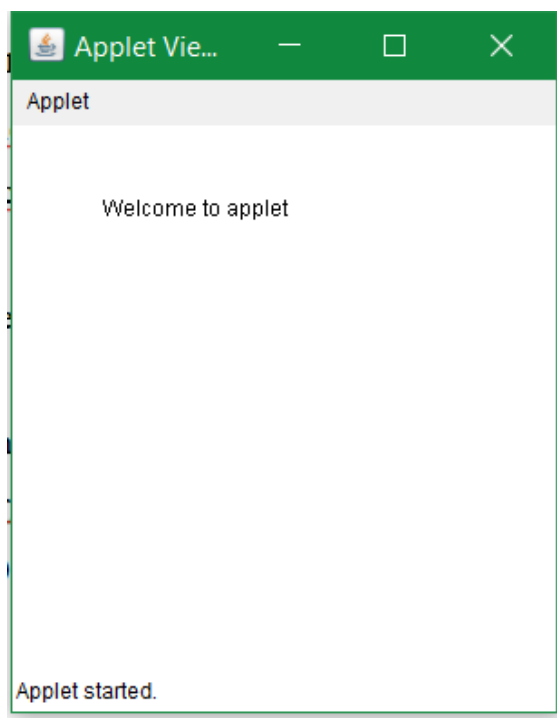
```
import java.applet.*;
```

```
import java.awt.*;

public class UseParam extends Applet
{
    public void paint(Graphics g)
    {
        String str=getParameter("msg");
        g.drawString(str,50, 50);
    }
}

/*<applet code="UseParam" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet> */
```

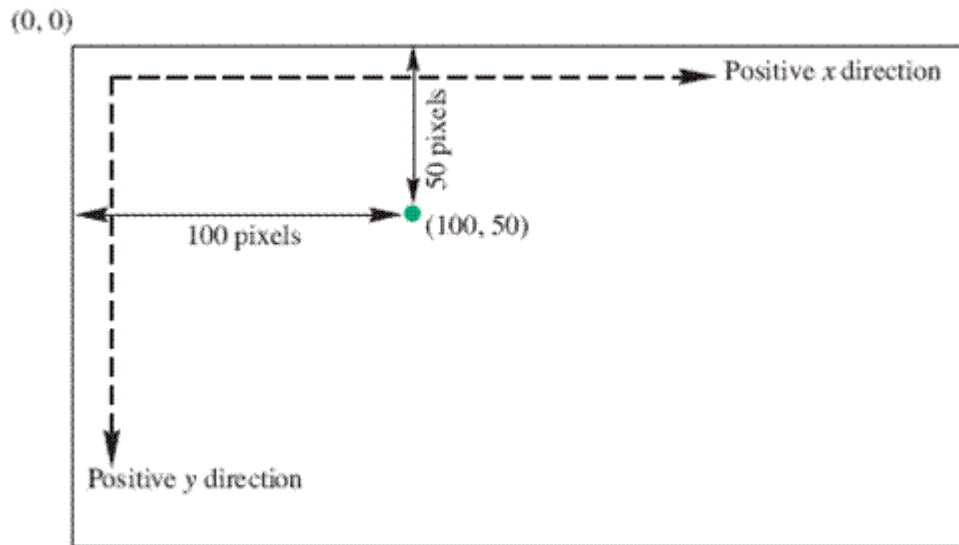
### **Output**



### **Working with Graphics**

One of the most important features of Java is its ability to draw graphics. We can write Java applets that draw lines, figures of different shapes, images and text in different fonts and styles. Every applet has its own area of the screen known as *canvas*, where it creates its display. A Java applet draws graphical image inside its space using the coordinate system as shown below.

Java coordinate system has the origin (0,0) in the upper left corner. Positive x values are to the right, and positive y values are to the bottom. The values of coordinates x and y are in pixels.



### **The Graphics Class**

Java's Graphics class includes methods for drawing many different shapes, from simple lines to polygons to text in a variety of fonts.

#### **Drawing methods of Graphics class**

<b>Method</b>	<b>Description</b>
clearRect()	Erases a rectangular area of the canvas.
copyArea()	Copies the rectangular area of the canvas to another area.
drawArc()	Draws a hollow arc
drawLine()	Draws a straight line
drawOval()	Draws a hollow oval
drawPolygon()	Draws a hollow polygon
drawRect()	Draws a hollow rectangle
drawRoundRect()	Draws a hollow rectangle with rounded corners
drawstring()	Displays a text string
fillArc()	Draws a filled arc
fillOval()	Draws a filled oval
fillPolygon()	Draws a filled polygon
fillRect()	Draws a filled rectangle
fillRoundRect()	Draws a filled rectangle with rounded corners
getColor()	Retrieves the current drawing color
getFont()	Retrieves the currently used font

setColor()	Sets the drawing color
getFont()	Sets the font

### Lines and Rectangles

The simplest shape that can be drawn with the Graphics is a line. The **drawLine()** method takes two pair of coordinates, (x1,y1) and (x2,y2) as arguments and draws a line between them.

For example the following code draws a straight line from (10,10) to (50,50):

```
g.drawLine(10,10,50,50);
```

g is the Graphics object passed to paint() method.

Similarly a rectangle can be drawn using **drawRect()** method. This method takes four arguments.

**The first two represent x and y coordinates from the top left corner of the rectangle and the remaining two represent the width and height of the rectangle.**

For example the following statement will draw a rectangle starting at (10,60) having a width of 40 pixels and a height of 30 pixels.

```
g.drawRect(10,60,40,30);
```

g is the Graphics object passed to paint() method.

The fillRect() method can be used to draw a solid box. This also takes four arguments (same as drawRect() method)

Example :: **g.fillRect(10,60,40,30);**

Rounded rectangles (Rectangles with rounded edges) can be created using the methods **drawRoundRect()** and **fillRoundRect()** . These methods are similar to drawRect() and fillRect() except that they take **two extra arguments representing the width and height of the angle of corners**. These extra parameters indicate how much of corners will be rounded.

Example:: **g.drawRoundRect(10,100,80,50,10,10);**

```
g.fillRoundRect(10,100,80,50,10,10);
```

### Circles and Ellipses

The Graphics class does not have any method for circles or ellipses. However **drawOval()** method can be used to draw a circle or ellipse. Ovals are just like rectangles with overly rounded corners. drawOval() method takes four arguments: **the first two represent the top left corner of the imaginary rectangle and the other two represent the width and height of the oval itself**. If width and height are the same, the oval becomes a circle. Ovals coordinates are actually the coordinates of an enclosing rectangle.

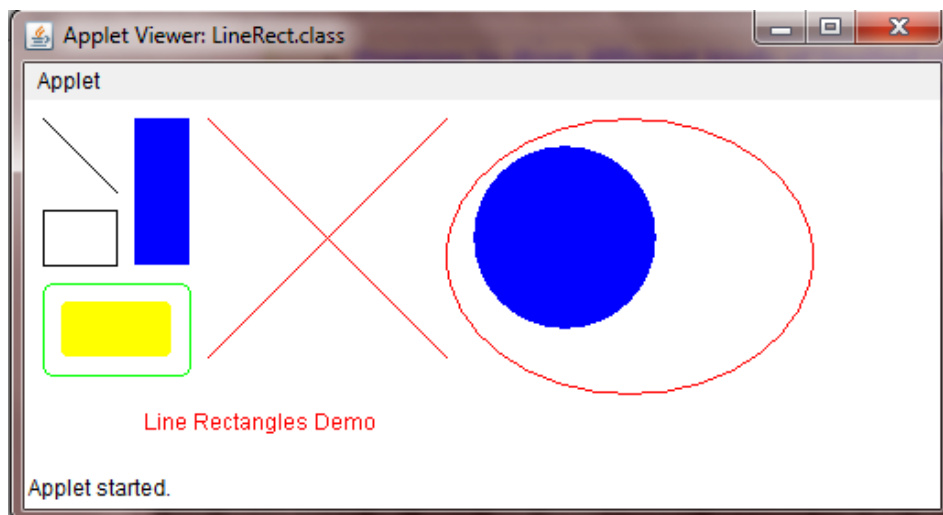
Like rectangle methods, drawOval() method draws outline of an oval, and the fillOval() method draws a solid Oval.

**Example 5: lines, rectangles, circles , ovals and rounded rectangles**

```

import java.awt.*;
import java.applet.*;
/*<applet code="LineRect " width=500 height=200></applet>*/
public class LineRect extends Applet
{
    public void paint(Graphics g)
    {
        g.drawLine(10,10,50,50);
        g.drawRect(10,60,40,30);
        g.setColor(Color.blue);
        g.fillRect(60,10,30,80);
        g.setColor(Color.green);
        g.drawRoundRect(10,100,80,50,10,10);
        g.setColor(Color.yellow);
        g.fillRoundRect(20,110,60,30,5,5);
        g.setColor(Color.red);
        g.drawLine(100,10,230,140);
        g.drawLine(100,140,230,10);
        g.drawString("Line Rectangles Demo",65,180);
        g.drawOval(230,10,200,150);
        g.setColor(Color.blue);
        g.fillOval(245,25,100,100);
    }
}

```

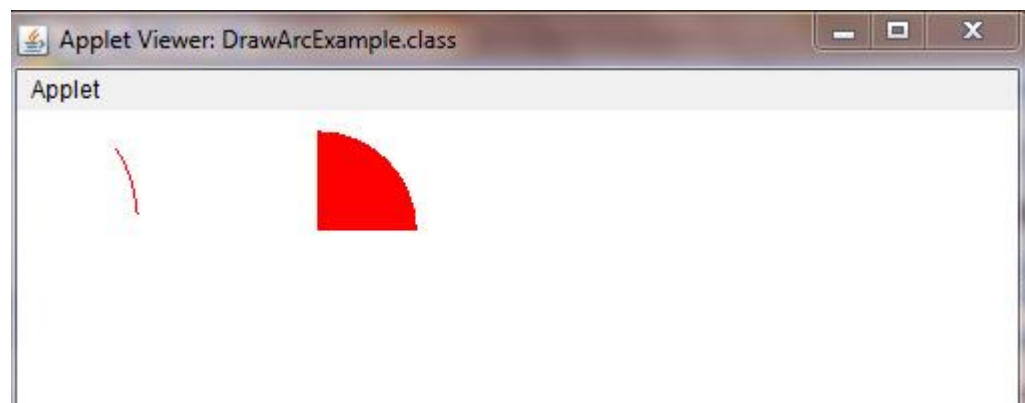
**Output****Drawing Arcs**

An arc is a part of an oval. The **drawArc()**, designed to draw arcs ,takes six arguments. **The first four are same arguments as that of drawOval() method and last two arguments represent the starting angle of the arc and the number of degrees(sweep angle) around the arc.** In drawing arcs , Java actually formulates the arc as an oval and then draws only a part of it as dictated by the last two arguments.

### ***Example 6: Drawing arcs***

```
/*<applet code="DrawArcExample.class" width=500 height=500>
</applet>*/
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
public class DrawArcExample extends Applet
{
    public void paint(Graphics g)
    {
        setForeground(Color.red); //set color to red
        g.drawArc(10,10,50,100,10,45);
        //this will draw an arc of width 50 & height 100 at (10,10)
        g.fillArc(100,10,100,100,0,90);
    }
}
```

### **Output**



### **Drawing polygons**

Polygons are shapes with many sides. A polygon may be considered a set of lines connected together. The end of first line is the beginning of the second line, and end of the second is the beginning of the third and so on. We can draw polygon with n sides using the **drawLine()** method n times in succession. We can draw polygons more conveniently using the **drawPolygon()** method of Graphics class.

This method takes three arguments.

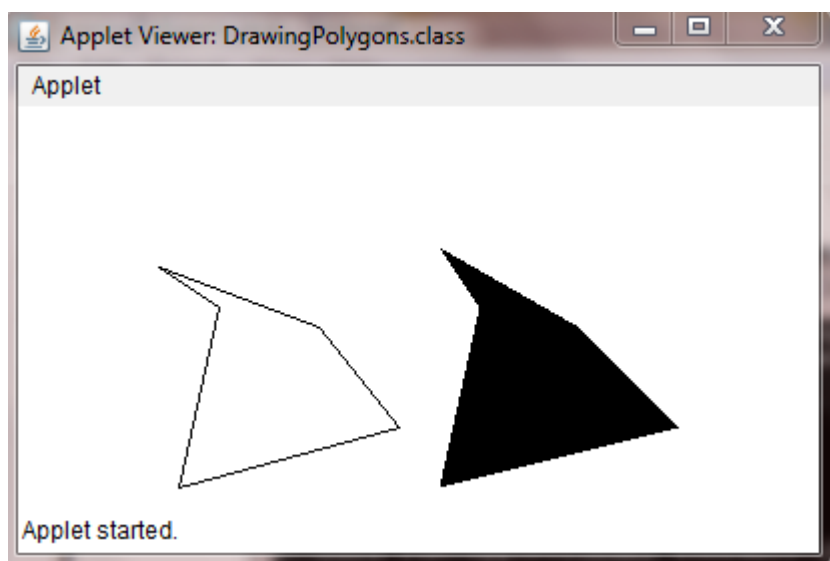
- An array of integers containing x coordinates.
- An array of integers containing y coordinates.
- An integer for the total number of points.

### **Example 7: Drawing Polygons**

```
import java.awt.*;
import java.applet.*;
/*<applet code="DrawingPolygons.class" width=400 height=200>
</applet> */
public class DrawingPolygons extends Applet
{
    public void paint(Graphics g)
    {
        int x[] = { 70, 150, 190, 80, 100 };
        int y[] = { 80, 110, 160, 190, 100 };
        g.drawPolygon (x, y, 5);

        int x1[] = { 210, 280, 330, 210, 230 };
        int y1[] = { 70, 110, 160, 190, 100 };
        g.fillPolygon (x1, y1, 5);
    }
}
```

### **Output**



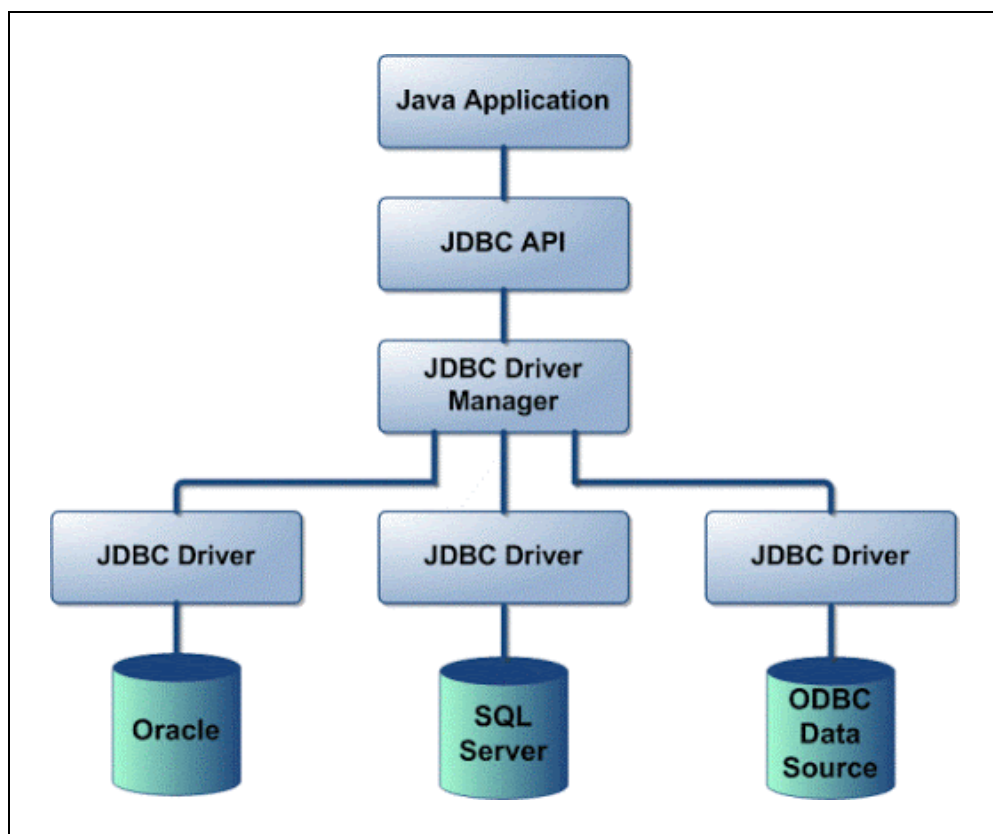
## **JDBC Architecture**

JDBC or Java Database Connectivity is a specification from Sun microsystems that provides a standard abstraction (that is API or Protocol) for java applications to communicate with various databases. It provides the language with java database connectivity standard. It is used to write programs required to access databases. JDBC along with the database driver is capable of accessing databases and spreadsheets. JDBC is an API (Application programming interface) which is used in java programming to interact with databases. The classes and interfaces of JDBC allow application to send request made by users to the specified database. Applications that are created using the JAVA need to interact with databases to store application-specific information. So, interacting with a database requires efficient database connectivity which can be achieved by using the ODBC (Open database connectivity) driver. This driver is used with JDBC to interact or communicate with various kinds of databases such as Oracle, MS Access, Mysql and SQL server database.

## **Architecture of JDBC**

The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers –

- JDBC API: This provides the application-to-JDBC Manager connection.
- JDBC Driver API: This supports the JDBC Manager-to-Driver Connection.



The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases. The JDBC driver manager ensures that the correct driver

is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases. Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application.

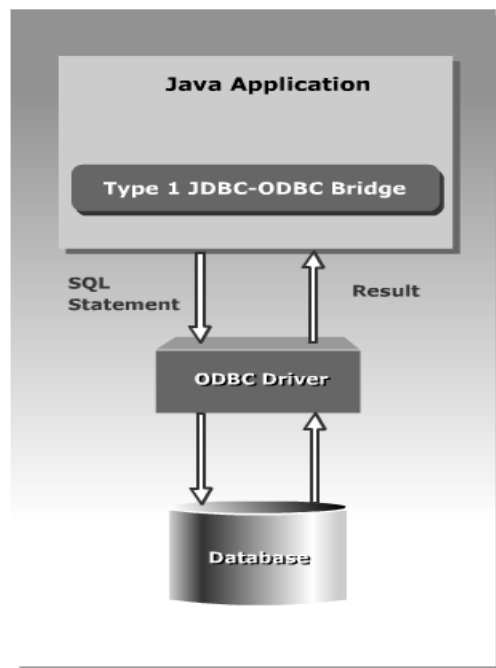
### **JDBC drivers**

JDBC drivers are client-side adapters that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

- Type-1 driver or JDBC-ODBC Bridge driver
- Type-2 driver or Native-API Partly-Java driver (partially java driver)
- Type-3 driver or JDBC-Net Pure-Java driver (fully java driver)
- Type-4 driver or Native-Protocol Pure-Java driver or Thin driver

### **JDBC-ODBC Bridge driver**

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases. As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured. The ODBC bridge driver is needed to be installed in individual client machines.

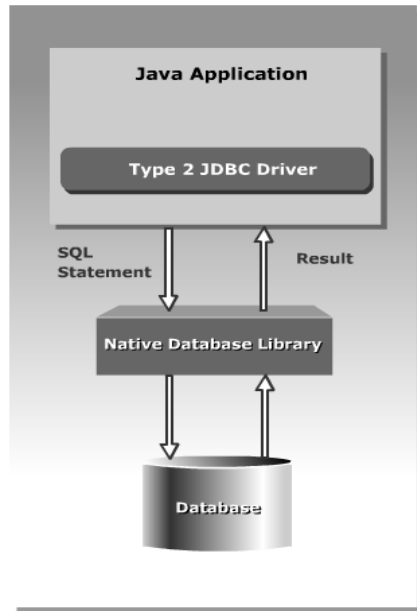


Type-1 driver isn't written in java, that's why it isn't a portable driver. This driver software is built-in with JDK so no need to install separately. It is a database independent driver.

### **Native-API Partly-Java driver**

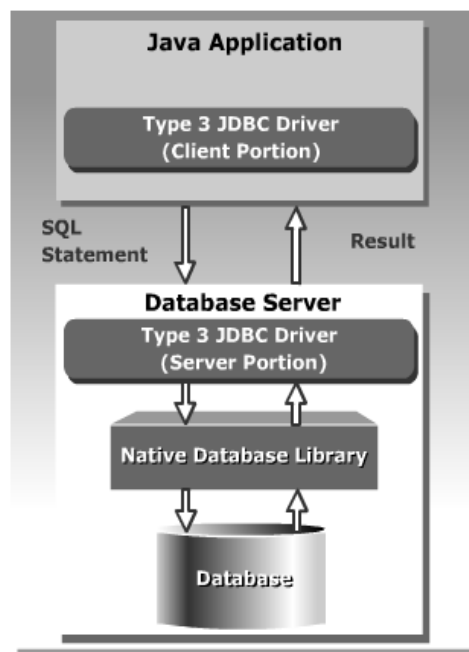
The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java. In order to

interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver. Driver needs to be installed separately in individual client machines. The Vendor client library needs to be installed on client machine. Type-2 driver isn't written in java, that's why it isn't a portable driver. It is a database dependent driver.



### **JDBC-Net Pure-Java driver**

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation. Type-3 drivers are fully written in Java, hence they are portable drivers.

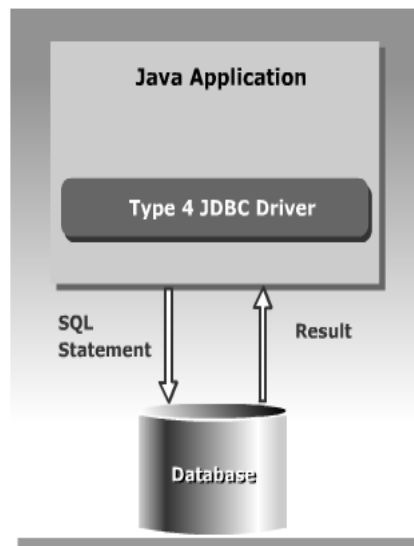


No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc. Network support is required on client machine. Maintenance

of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier. Switch facility to switch over from one database to another database.

### **Native-Protocol Pure-Java driver**

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language. It does not require any native database library that is why it is also known as Thin Driver.



This driver does not require any native library and Middleware server, so no client-side or server-side installation. It is fully written in Java language, hence they are portable drivers.

### **Creating JDBC Applications**

The steps to create JDBC application are:

- Load a driver
- Connect to a database
- Create and execute JDBC statements
- Handle SQL exceptions

### **Loading a Driver**

Programmatically:

- Using the `forName()` method
- Using the `registerDriver()` method

Manually:

- By setting system property

### **DriverManager class**

In Java, the DriverManager class is an interface between the User and the Driver. This class is used to have a watch on driver which is been used for establishing the connection between a database and a driver. The DriverManager class has a list of Driver class which are registered and are called

as **DriverManager.registerDriver()**.

### **Connecting to a Database**

The DriverManager class provides the getConnection() method to create a Connection object.

The getConnection() method has the following three forms:

- Connection getConnection (String <url>)
- Connection getConnection (String <url>, String <username>, String <password>)
- Connection getConnection (String <url>, Properties <properties>)

### **Creating and Executing JDBC Statements**

The Connection object provides the createStatement() method to create a Statement object. You can use static SQL statements to send requests to a database to retrieve results. The Statement interface contains the following methods to send static SQL statements to a database:

- ResultSet executeQuery(String str)
- int executeUpdate(String str)
- boolean execute(String str)

### **Statement interface**

In Java, The Statement interface is used for executing queries using the database. This interface is a factory of ResultSet. It is used to get the Object of ResultSet. Methods of this interface are given below.

S.No.	Method	Description
1	public ResultSet executeQuery(String sql)	It is used for executing the SELECT query
2	public int executeUpdate(String sql)	It is used for executing any specified query
3	public boolean execute(String sql)	It is used when multiple results are required.
4	public int[] executeBatch()	It is used for executing the batch of commands.

### **ResultSet interface**

In Java, the ResultSet Interface is used for maintaining the pointer to a row of a table. In starting the pointer is before the first row. The object can be moved forward as well as backward direction in createStatement.

### **PreparedStatement interface**