Java Programming-V

By:- Jagdish Chandra Pandey

H.O.D(I.T.), G.P. Dwarahat/Officiating Principal, G.P.Chaunaliya

Exception Handling in Java

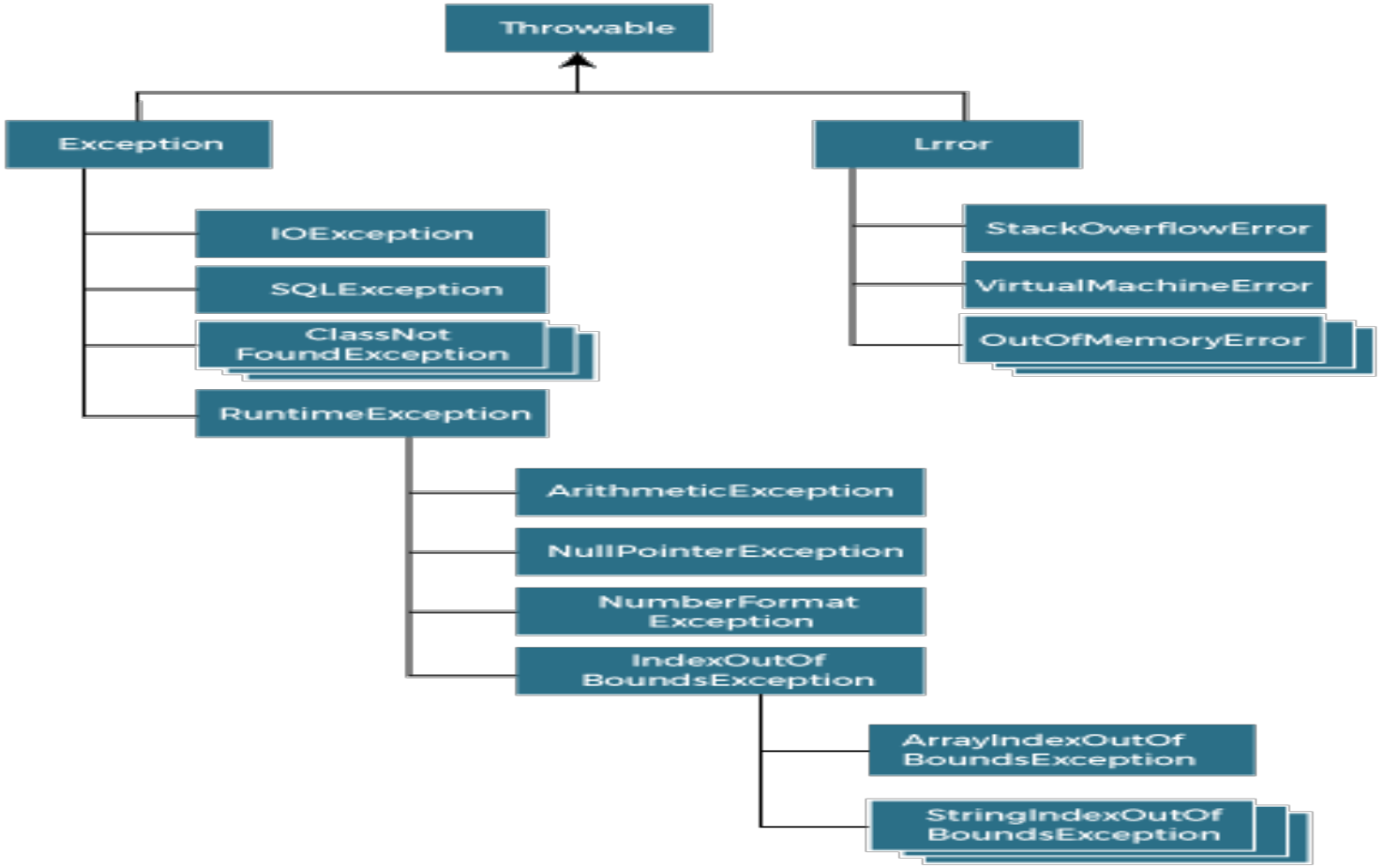
- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

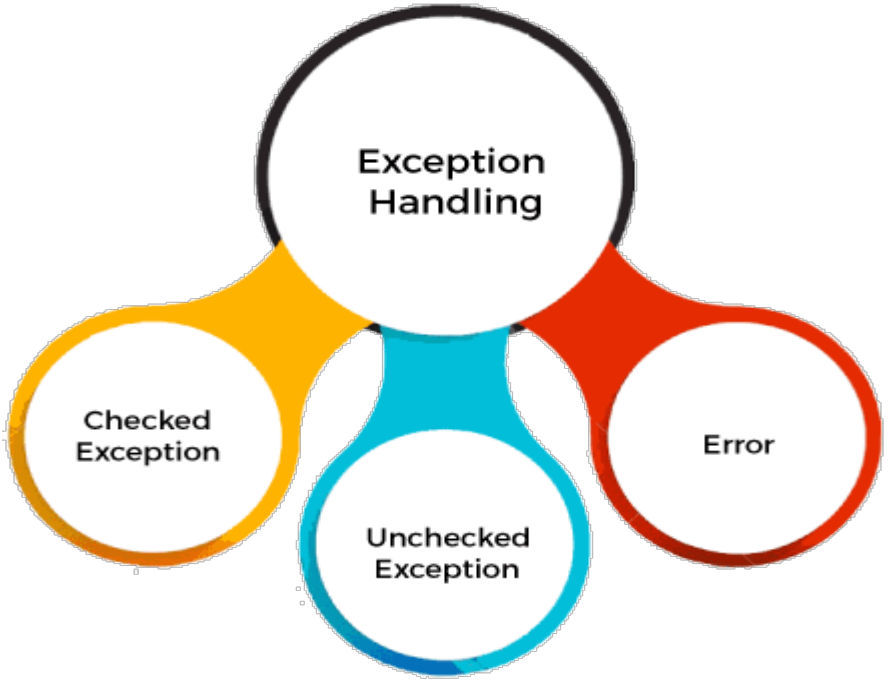
Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

1. Checked Exception
2. Unchecked Exception
3. Error



Types of Java Exceptions

1) Checked Exception:-

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception:-

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error:-

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each:-

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

```
public class JavaExceptionExample
{
public static void main(String args[])
{
try
{
//code that may raise exception
 int data=100/0;
 }
catch(ArithmeticException e)
{
System.out.println(e);
}
//rest code of the program
System.out.println("rest of the code...");
 }
}
```

Output:

- Exception in thread main java.lang.ArithmeticException:/ by zero rest of the code...

Java try-catch block

- Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

- Java try block must be followed by either catch or finally block.

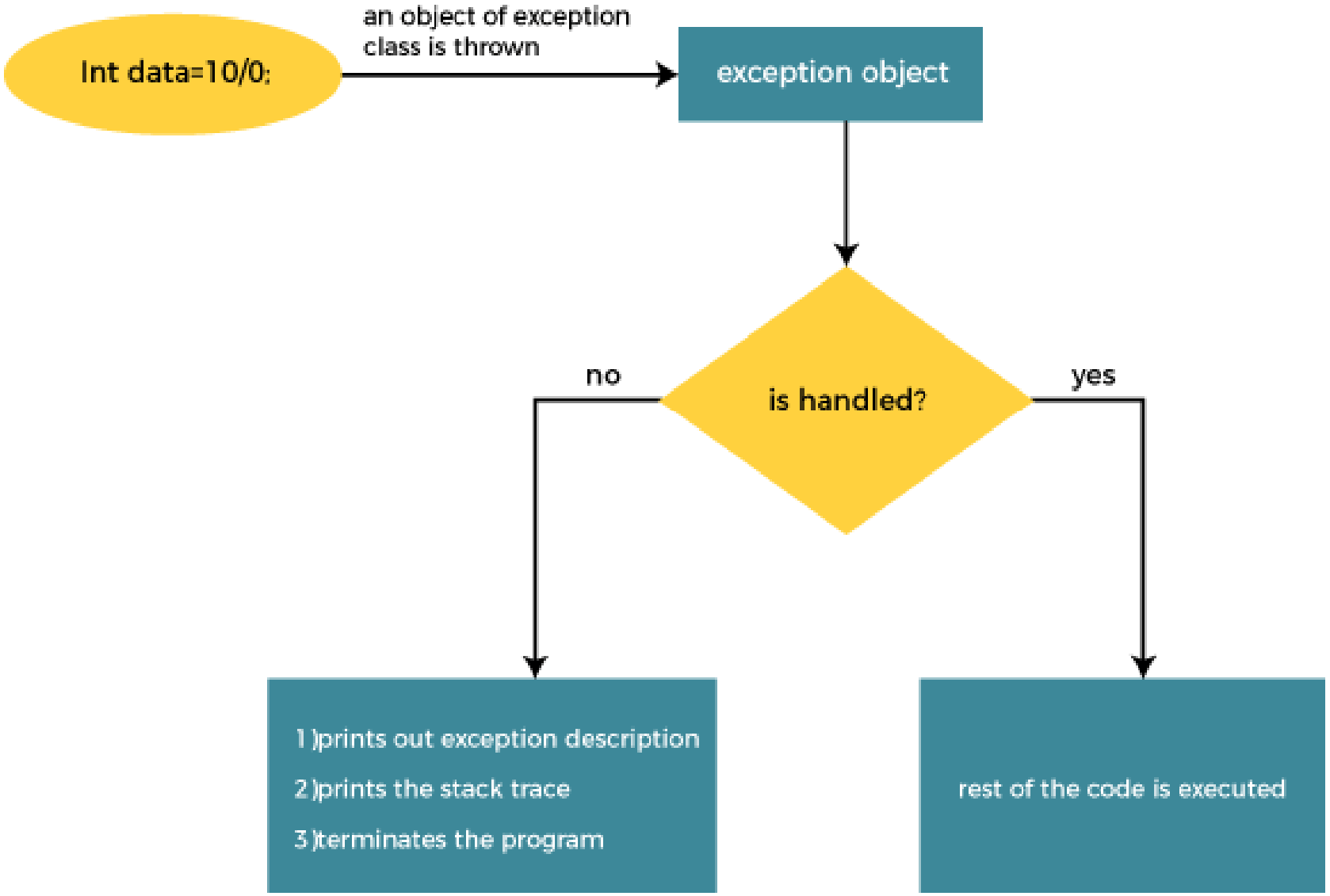
- Syntax of Java try-catch

```
try
  {
   //code that may throw an exception
   }
    catch(Exception_class_Name ref){}
```

- Syntax of try-finally block

```
try
 {
   //code that may throw an exception
 }
 finally{}
```

Internal Working of Java try-catch block



Internal Working of Java try-catch block

- The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:-

1. Prints out exception description.
2. Prints the stack trace (Hierarchy of methods where the exception occurred).
3. Causes the program to terminate.

- But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

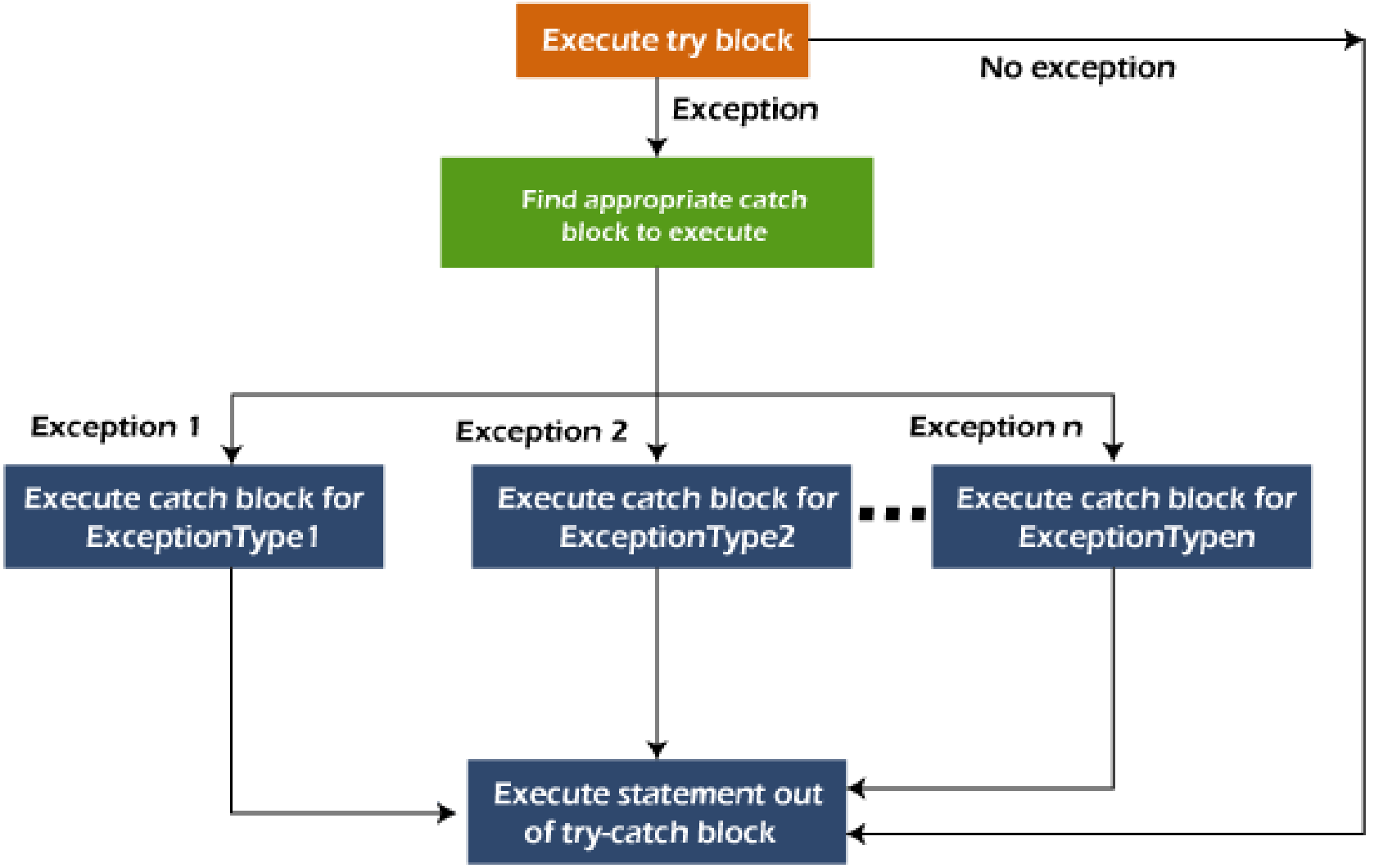
Java Multi-catch block

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

- At a time only one exception occurs and at a time only one catch block is executed.

- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Java Multi-catch block



Example Java Multi-catch block

```
public class MultipleCatchBlock1
 {
    public static void main(String[] args)
{
 try
{
            int a[]=new int[5];
         a[5]=30/0;
}
  catch(ArithmeticException e)
  {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
   }
}
```

- **Output:** Arithmetic Exception occurs rest of the code

Java Nested try block

- In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

- For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Syntax:Java Nested try block

```
//main try block
try
{
   statement 1;
   statement 2;
//try catch block within another try block
   try
   {
      statement 3;
      statement 4;
//try catch block within nested try block
      try
      {
         statement 5;
         statement 6;
    }
      catch(Exception e2)
      {
//exception message
      }

   }
   catch(Exception e1)
   {
//exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
```

Example:Java Nested try block

```
public class NestedTryBlock2 {

   public static void main(String args[])
   {
      // outer (main) try block
      try {

         //inner try block 1
         try {

            // inner try block 2
            try {
               int arr[] = { 1, 2, 3, 4 };

               //printing the array element out of its bounds
               System.out.println(arr[10]);
            }

            // to handles ArithmeticException
            catch (ArithmeticException e) {
               System.out.println("Arithmetic exception");
               System.out.println(" inner try block 2");
            }
         }

         // to handle ArithmeticException
         catch (ArithmeticException e) {
            System.out.println("Arithmetic exception");
            System.out.println("inner try block 1");
         }
      }

      // to handle ArrayIndexOutOfBoundsException
      catch (ArrayIndexOutOfBoundsException e4) {
         System.out.print(e4);
         System.out.println(" outer (main) try block");
      }
      catch (Exception e5) {
         System.out.print("Exception");
         System.out.println(" handled in main try-block");
      }
   }
}
```

Output:-

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock2
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 outer
 (main) try block
```

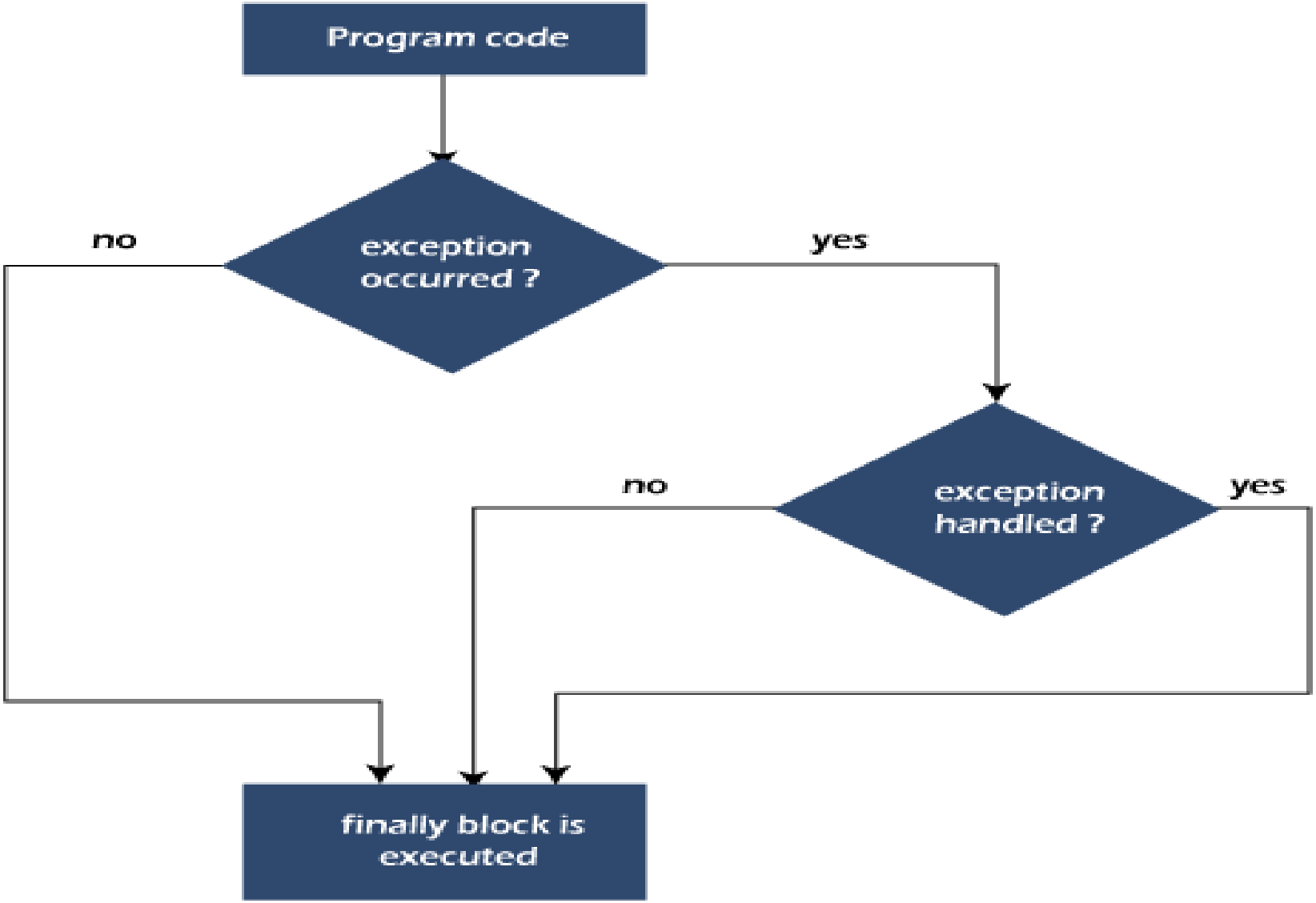
Java finally block

- **Java finally block** is a block used to execute important code such as closing the connection, etc.

- Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.

Flowchart of finally block



Example of finally block

```
public class TestFinallyBlock1
{
    public static void main(String args[])
{
    try {

     System.out.println("Inside the try block");
     //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    //cannot handle Arithmetic type exception
    //can only accept Null Pointer type exception
    catch(NullPointerException e)
  {
     System.out.println(e);
    }
    //executes regardless of exception occured or not
    finally {
     System.out.println("finally block is always executed");

  }
    System.out.println("rest of the code...");
    }
   }
```

Output:-

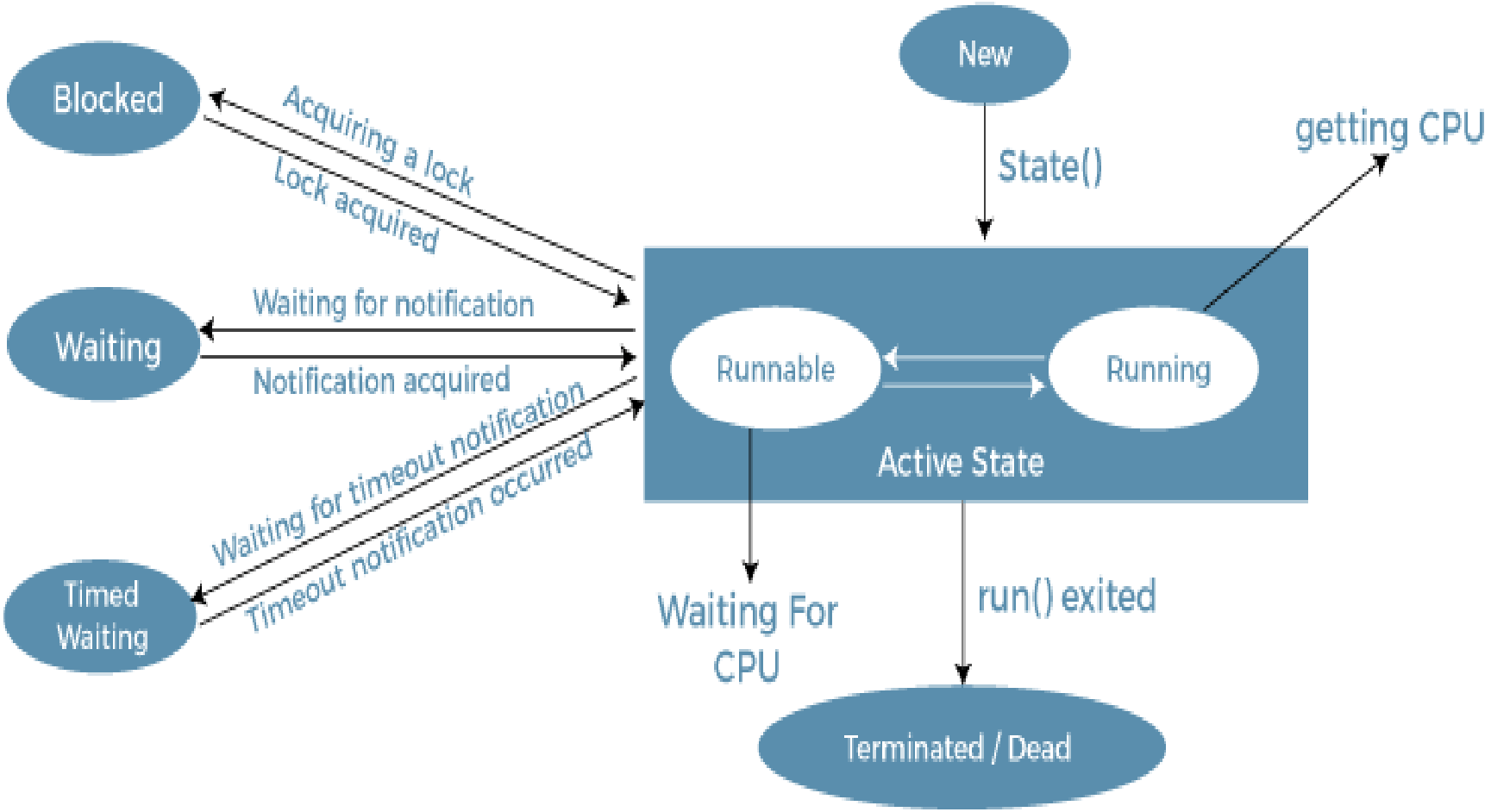
```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Thread and Multithreading in Java

- A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process). You can think of a lightweight process as a virtual CPU that executes code or system calls.
- Multithreaded programs contain two or more threads that can run concurrently and each thread defines a separate path of execution. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

Life cycle of a Thread (Thread States)



Life Cycle of a Thread

Explanation of Different Thread States

- **New:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

- **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**.

1. **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
   A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time.
   Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

2. **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Explanation of Different Thread States

- **Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

- **Timed Waiting:** Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever.

- **Terminated:** A thread reaches the termination state because of the following reasons:-

1. When a thread has finished its job, then it exists or terminates normally.

2. **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

Java Thread Class

- Thread class, along with its companion interface Runnable will be used to create and run threads for utilizing Multithreading feature of Java.

- It provides constructors and methods to support multithreading. It extends object class and implements Runnable interface.

public class Thread extends Object implements Runnable

Thread Class Priority Constants

Field	Description
MAX_PRIORITY	It represents the maximum priority that a thread can have.
MIN_PRIORITY	It represents the minimum priority that a thread can have.
NORM_PRIORITY	It represents the default priority that a thread can have.

Constructors of Thread class

1. **Thread**()
2. **Thread**(String str)
3. **Thread**(Runnable r)
4. **Thread**(Runnable r, String str)
5. **Thread**(**ThreadGroup** group, **Runnable** target)
6. **Thread**(**ThreadGroup** group, **Runnable** target, String name)
7. **Thread**(**ThreadGroup** group, **Runnable** target, String name, long stackSize)
8. **Thread**(**ThreadGroup** group, **String** name)

Thread Class Methods

Method	Description
setName()	to give thread a name
getName()	return thread's name
getPriority()	return thread's priority
isAlive()	checks if thread is still running or not
join()	Wait for a thread to end
run()	Entry point for a thread
sleep()	suspend thread for a specified time
start()	start a thread by calling run() method
activeCount()	Returns an estimate of the number of active threads in the current thread's thread group and its subgroups.
checkAccess()	Determines if the currently running thread has permission to modify this thread.
currentThread()	Returns a reference to the currently executing thread object.
dumpStack()	Prints a stack trace of the current thread to the standard error stream.
getId()	Returns the identifier of this Thread.

Creating a thread in Java

- Java defines two ways by which a thread can be created-:

1. By implementing the **Runnable** interface.

2. By extending the **Thread** class.

Implementing the Runnable Interface

- The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface, the class needs to implement the run() method.

- Run Method Syntax:-

  public void run()

- It introduces a concurrent thread into your program. This thread will end when run() method terminates.

- You must specify the code that your thread will execute inside run() method.

- run() method can call other methods, can use other classes and declare variables just like any other normal method.

Implementing the Runnable Interface

```
class MyThread implements Runnable
{
 public void run()
 {
System.out.println("concurrent thread started running..");
   }
}
class MyThreadDemo
{
   public static void main(String args[])
   {
      MyThread mt = new MyThread();
      Thread t = new Thread(mt);
      t.start();
   }
}
```

To call the run()method, start() method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

Output:-

concurrent thread started running..

Extending Thread class

- This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class. The extending class must override run() method which is the entry point of new thread.

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("concurrent thread started running..");
    }
}

classMyThreadDemo
{
    public static void main(String args[])
    {
        MyThread mt = new  MyThread();
        mt.start();
    }
}
```

Output:- concurrent thread started running..

Java Synchronization

- At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

Syntax:-

```
synchronized (object)
{
//statement to be synchronized
}
```

Example : implementation of synchronized method

```
class First
{
 synchronized public void display(String msg)
 {
  System.out.print ("["+msg);
  try
  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }
  System.out.println ("]");
 }
}

class Second extends Thread
{
 String msg;
 First fobj;
 Second (First fp,String str)
 {
  fobj = fp;
  msg = str;
  start();
 }
 public void run()
 {
  fobj.display(msg);
 }
}
```

```
public class MyThread
{
 public static void main (String[] args)
 {
  First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second(fnew,"new");
  Second ss2 = new Second(fnew, "programmer");
 }
}
```

Output:-

[welcome]
[new]
[programmer]

In the above program, object **fnew** of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(void **display**).

To synchronize above program,we must *synchronize* access to theshared **display()** method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display() method.

Using Synchronized block

- If want to synchronize access to an object of a class or only a part of a method to be synchronized then we can use synchronized block for it. It is capable to make any part of the object and method synchronized.

```
class First
{
 public void display(String msg)
 {
  System.out.print ("["+msg);
  try
  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }
  System.out.println ("]");
 }
}

class Second extends Thread
{
 String msg;
 First fobj;
 Second (First fp,String str)
 {
  fobj = fp;
  msg = str;
  start();
 }
 public void run()
 {
  synchronized(fobj)      //Synchronized block
  {
   fobj.display(msg);
  }
 }
}
public class MyThread
{
 public static void main (String[] args)
 {
  First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second (fnew,"new");
  Second ss2 = new Second(fnew, "programmer");
 }
}
```

Output:-

```
[welcome]
 [new]
 [programmer]
```

Difference between synchronized keyword and synchronized block

- When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution.

- synchronized block acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

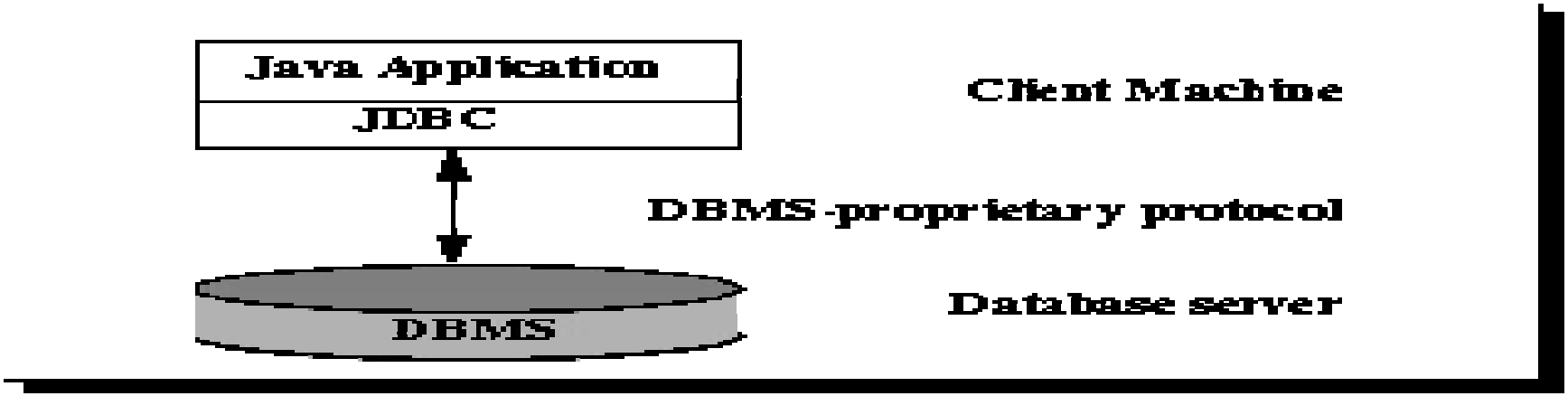
- In Java, synchronized keyword causes a performance cost. A synchronized method in Java is very slow and can degrade performance. So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

Introduction to JDBC

- JDBC is an API(Application programming interface) which is used in java programming to interact with databases.
- 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.

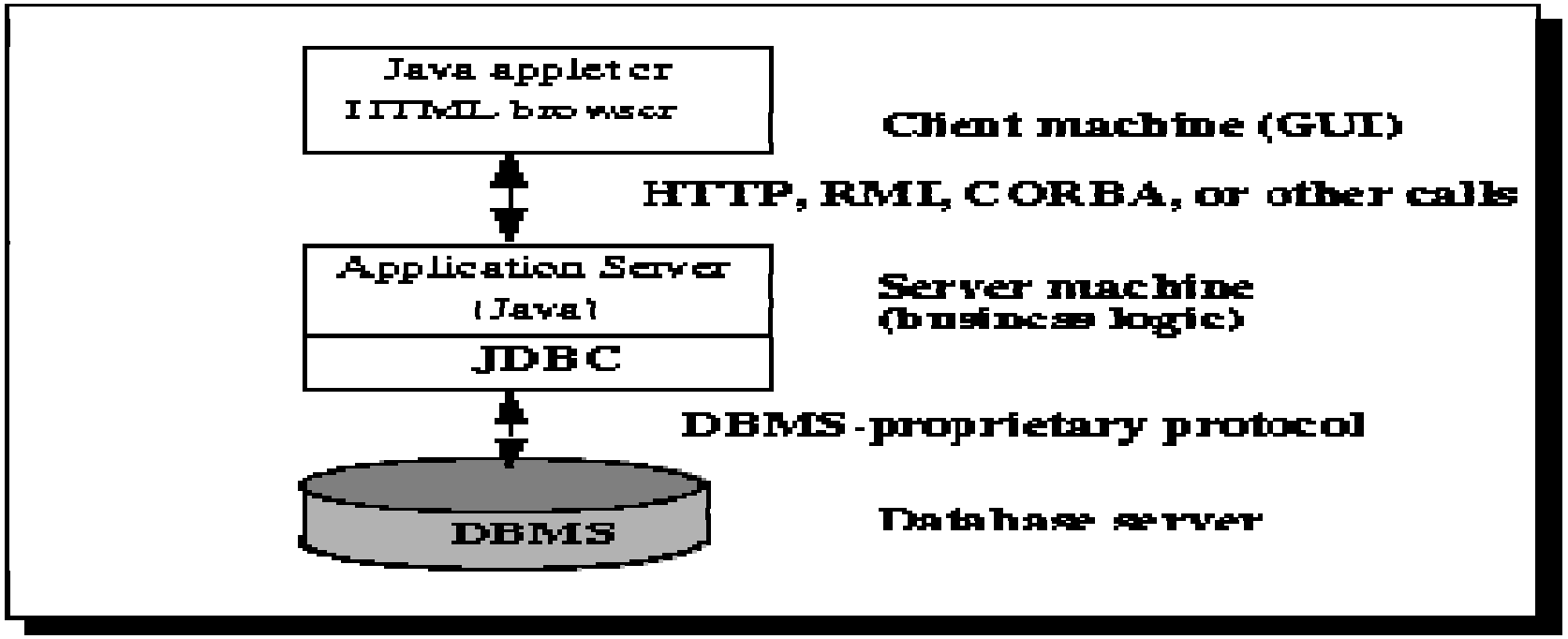
JDBC Architecture

- 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.



JDBC Architecture

- 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.



JDBC Driver

- JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:-

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java 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.

- Oracle does not support the JDBC-ODBC Bridge from Java 8.

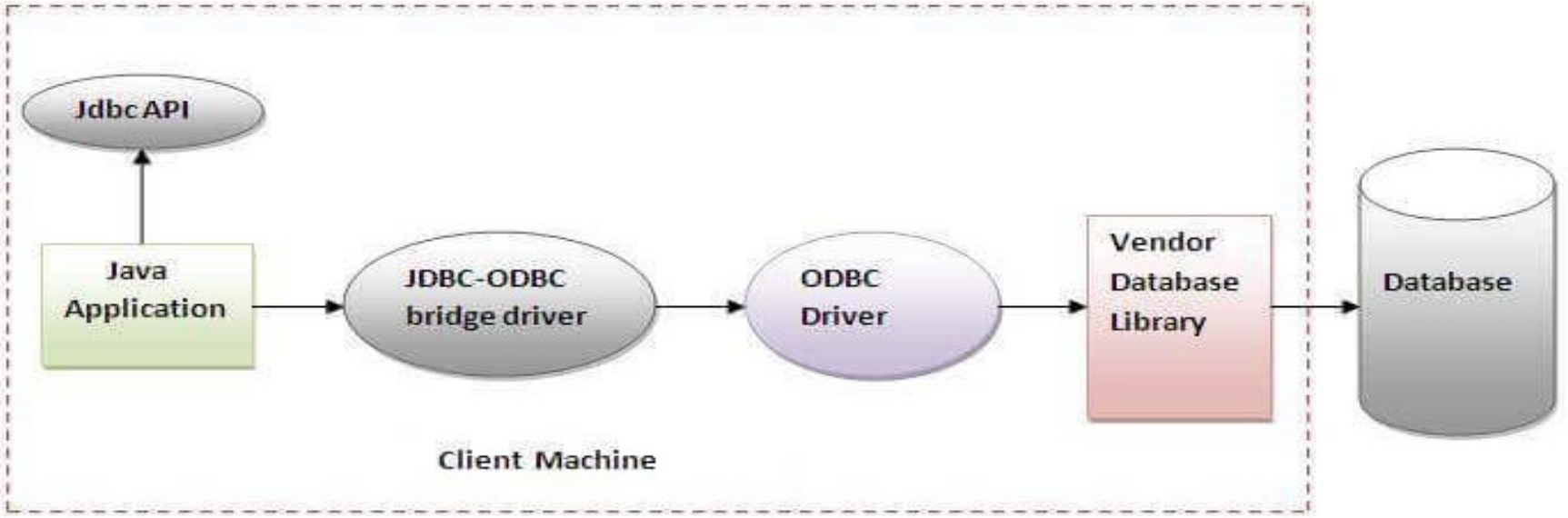


Figure- JDBC-ODBC Bridge Driver

JDBC-ODBC bridge driver

- Advantages:-

1. easy to use.
2. can be easily connected to any database.

- Disadvantages:-

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

Native-API 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.

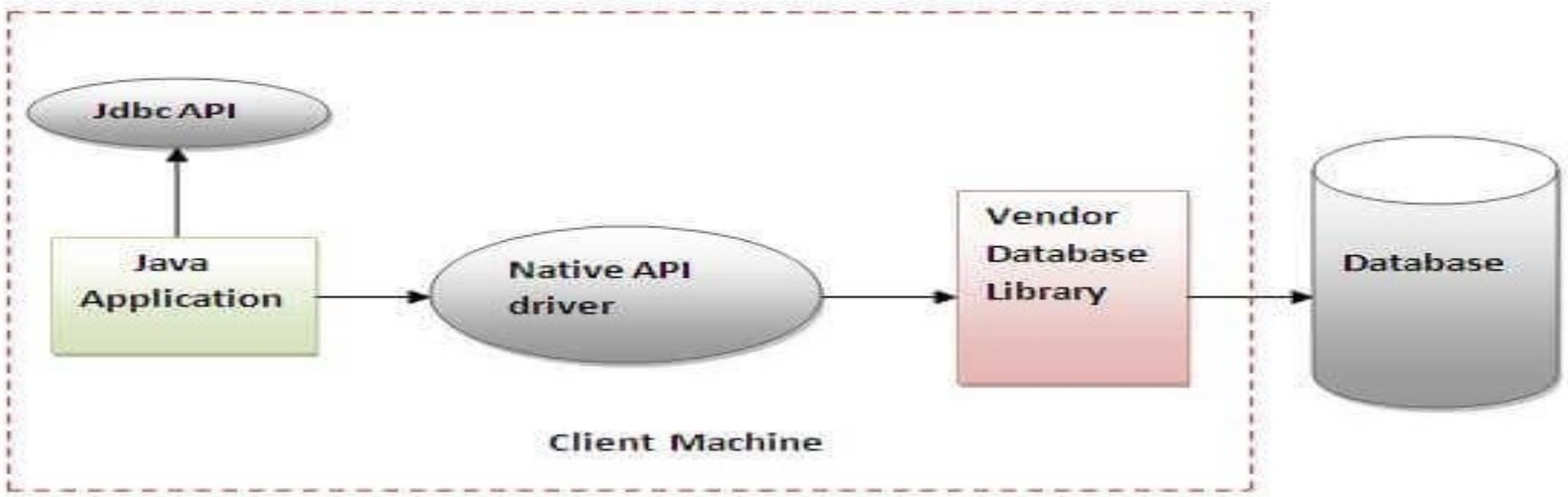


Figure- Native API Driver

Native-API driver

- Advantage:-

1. performance upgraded than JDBC-ODBC bridge driver.

- Disadvantage:-

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

Network Protocol 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.

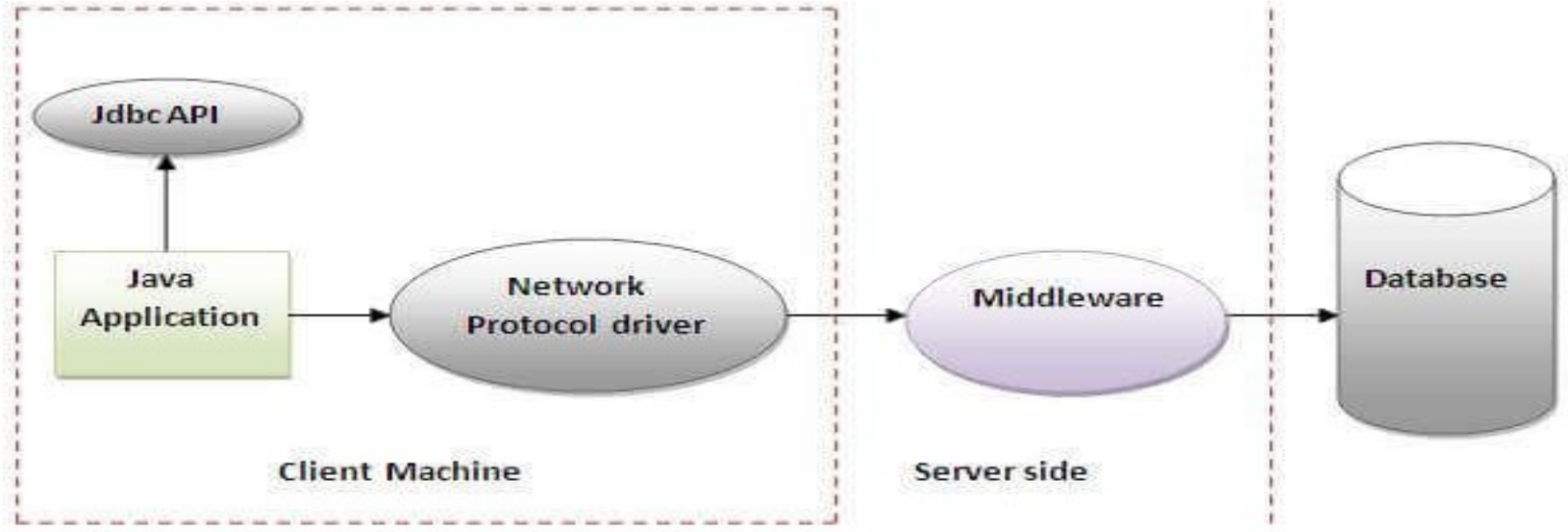


Figure- Network Protocol Driver

Network Protocol driver

- Advantages:-

1. No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

- Disadvantages:-

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

Thin 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.

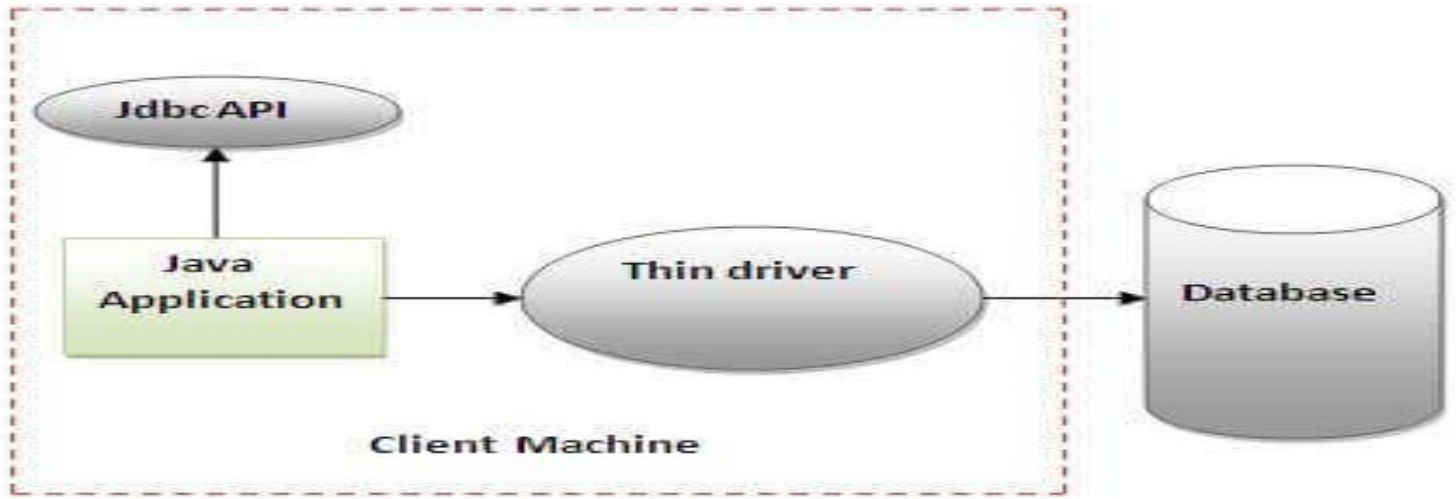


Figure- Thin Driver

Thin driver

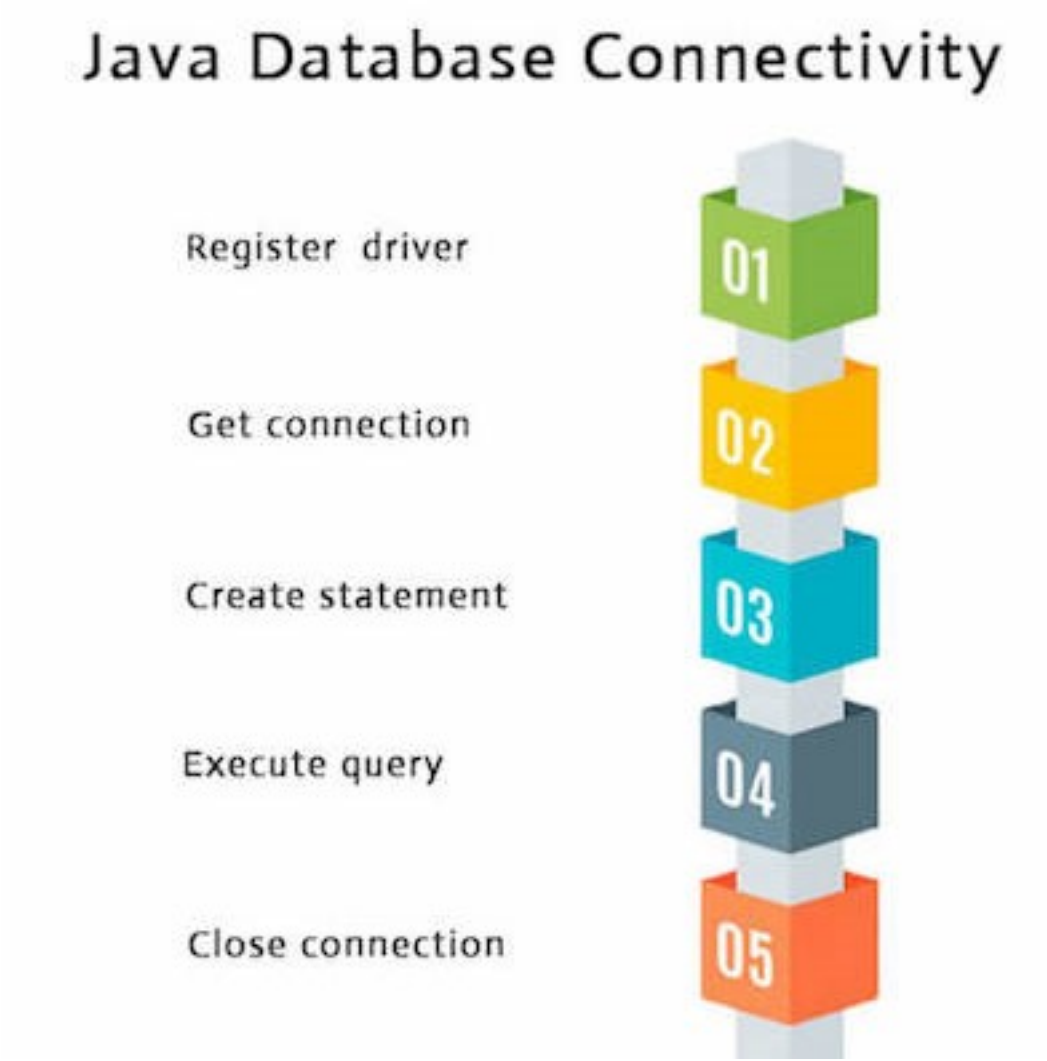
- Advantage:-

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

- Disadvantage:-

1. Drivers depend on the Database.

Java Database Connectivity with 5 Steps



Register the driver class

1. The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

2. Syntax of forName() method:-

```
public static void forName(String className)throws ClassNotFoundException
```

3. Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

4. Example to register the OracleDriver class:- Here, Java program is loading oracle driver to esteblish database connection.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Create the connection object

1. The **getConnection()** method of DriverManager class is used to establish connection with the database.

2. Syntax of getConnection() method:-

   1) **public static** Connection getConnection(String url)**throws** SQLException

   2) **public static** Connection getConnection(String url,String name,String password) **throws** SQLException

3. Example to establish connection with the Oracle database:-

```
Connection con=DriverManager.getConnection(  "jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

Create the Statement object

- The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

- Syntax of createStatement() method:-

```
public Statement createStatement()throws SQLException
```

- Example to create the statement object:-

```
Statement stmt=con.createStatement();
```

Execute the query

- The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

- Syntax of executeQuery() method:-

```
public ResultSet executeQuery(String sql)throws SQLException
```

- Example to execute query:-

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
{
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Close the connection object

- By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

- Syntax of close() method:-

  public void close()**throws** SQLException

- Example to close connection:-

  con.close();

- Since Java 7, JDBC has ability to use try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement. It avoids explicit connection closing step.

Example to Connect Java Application with Oracle database

- In this example, we are connecting to an Oracle database and getting data from **emp** table. Here, **system** and **oracle** are the username and password of the Oracle database.

```
import java.sql.*;
class OracleCon
{
public static void main(String args[])
{
Try
{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");
  //step2 create  the connection object
Connection con=DriverManager.getConnection(  "jdbc:oracle:thin:@localhost:1521:xe","system","or
      acle")
//step3 create the statement object
Statement stmt=con.createStatement();
//step4 execute query
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
//step5 close the connection object
con.close();
catch(Exception e){ System.out.println(e);
}
}
}
```


"THANK
YOU"