Core Java Programming

Presented By:

Corporate Trainer

Mallikarjuna G D

gdmallikarjuna@gmail.com

CONTENTS

- ✓ LAMDA
- ✓ FUNCTIONAL INTERFACE
- ✓ STREAMS API
- ✓ JDBC
- ✓ CAPSTONE PROJECT

LAMDA EXPORESSIONS

- It is a new feature introduced in Java 8.0
- It is a anonymous/nameless function which has no name, no modifier, no return type
- It is shorthand representation with interface without implementations
- Higher order function/closure

Advantages

Saves lots of code to be written

Enables functional programming

useful for collection library

Improves modularity

Disadvantages

Beginners difficult to understand

Maintenance is difficult

LAMDA EXPORESSIONS

- Allows writing of readable, maintainable, and concise code
- Helps to write API easy and effective
- To enable parallel processing
- Improves the productivity of developers

variable = function (); // you can assign function to a variable
function(funtion1()); // function can pass to another function

How to write Lambda expression :

```
greetingFun = public void greet(){
    System.out.println("Hello World");
  }
 greetingFun = () ->System.out.println("Hello world");

  doubleNumberFun = public int double(int a){
          return a*2;
    }
 doubleNumberFun = (int a) ->a*2;
```

```
addFun = public int add(int a, int b){
    return a+b;
  }
 addFun = (int a, int b) ->a+b;

   divFun = public int divide(int a, int b){
    return a/b;
  }
divFun = (int a, int b) ->{
    if (b == 0) return 0;
    return a/b;
};
stringLengthCount = public int strLen(String s){
    return s.length();
  }
```

```
stringLengthCount = public int strLen(String s){
    return s.length();
  }
 stringLengthCount = (String s) ->s.length();

public int square(int n){
 return n*n;
}
n->n*n;
```

conclusions

1. any number of arguments
2. for one argument parenthesis are optional
3. guess datatype for arguments passing
4. curly braces are mandatory more than one statements

How to call/invoke expression:

Functional Interface::

Any method which has single abstract methods are referred as Functional interface

Single Abstract Method

Runnable ->run()

Callable -> call()

Comparable ->compareTo()

To call Lambda we require a functional interface

How to call/invoke expression:

Functional Interface::

NOTE:

Lamda is used to implement the functional interface is very simple and short

Before java 1.8 interfaces should contain only abstract methods

In java 1.8 onwards we can write default and static methods in interfaces

```
static void m1() {}
default void m1(){}
```

```
@FunctionalInteface -annotation
@FunctionalInteface
interface A{
 public void m1();
}
@FunctionalInteface  //VALID
interface B extends A{


}
@FunctionalInteface  //VALID OVERRIDING
interface C extends A{
  public void m1();
}
```

```
@FunctionalInteface  //INVALID
interface D extends A{
  public void m2();
}
interface E extends A{
  public void m3();
}
```

```
interface InterA
{
  public void m1();
}
class Demo implements InterA{
  public void m1(){
   System.out.println("welcome");
  }
}
class TestA{
 public static void main(String args[]){
InterA a = new Demo();
          a.m1();
 }
}
```

scenario: when one argument passed

```
Class TestA{
 public static void main(String args[]){
    InterA a =()->System.out.println("welcome");
            a.m1();  }
}
```

scenario 2: when two arguments passed

```
interface Iadd{
  public int add(int a, int b);
}
Class TestA{
 public static void main(String args[]){
Iadd a =(a,b)->System.out.println("add"+(a+b));
            a.add();
  }
}
```

Scenario 3: when use default and static method used in functionainterface

```
interface Iadd{
  public int add(int a, int b);
  default void m1(){
   System.out.println("welcome to m1");
  }
  static void m2(){
   System.out.println("welcome to m2");
  }
}
Class TestA{
 public static void main(String args[]){
Iadd a =(a,b)->System.out.println("add"+(a+b));
            a.add();
            a.m1();
            Iadd.m2(); //static will be reference of interface
 } }
```

Scenario 4: when we use the square of a number

```
interface Inter{
 public int square(int n);
}
class Test{
  public static void main(String args[[]){
  Inter i = n -> n*n;
  System.out.println("the square of number :20 is"+i.square(20));
  System.out.println("the square of number :30 is"+ i.square(30));
  }
 }
```

Note: Inter i = n -> {n*n}; //INVALID - Return statement is compulsary to make it valid when u have curly braces

Inter i = n -> { return n*n;}; //valid return , semicolon to statement and curly braces required

Scenario 5: Thread Implementations

```
Runnable myThreadEx = ()->{
 for(int i=0;i<10;i++)
  {
    System.out.println("welcome to child thread");
  }
}

 button.addActionListner( ae->{
          System.out.println("Button Click");
                    JOptionPane.showMessageDialog(null,"I am here");
           }
  );
```

Scenario 5: Thread Implementations

```
Runnable myThreadEx = ()->{
 for(int i=0;i<10;i++)
  {
    System.out.println("welcome to child thread");
  }
}

 button.addActionListner( ae->{
          System.out.println("Button Click");
                    JOptionPane.showMessageDialog(null,"I am here");
           }
  );
```

The predefined functional interface supports you build many situations to create your own custom interface. The java gives a functional interface to express the lambda expressions. These are available in java.util.function package.

Predicate Interface : boolean result

It is a interface is boolean-valued function with one argument. This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.
It is conditional check.
Predicate<T>
- boolean test(T t)

```
Predicate<String> isStrLength = str->str.length > 5 ;
        isStrLength.test("Mallikarjuna");  // returns true
        isStrLength.test("Arun"); // returns false
```

Function Interface :

It is a functional interface that accepts anything and returns any.

```
Function<T,R>
    - R apply(T ,R)

Function<Integer, Integer>  square= a->a*a ;
        square.apply(10);  // returns 100

Function<String,Integer> len = str->str.length();
System.out.println(len.apply("Mallikarjuna"));
```

Consumer Interface :

It is a functional interface that accepts anything but no return type.

Consumer<T>

- *void accept(T)*

```
class Student{
String collegeName;
public Student() {}
public String getCollegeName() {
return collegeName;
}
public void setCollegeName(String collegeName) {
this.collegeName = collegeName;
}

Student std=new Student();
Consumer<Student> con = a->a.setCollegeName("JNNCE");
con.accept(std);
System.out.println(std.getCollegeName());
```

Supplier Interface :

It is a functional interface that returns but nothing accept as input.

Supplier<T>

- *T get( )*

```
Supplier<Double> sup = ()->Math.random();
System.out.println(sup.get());

static String prodName ="TV";
Supplier<String> e = ()->prodName;
System.out.println(e.get());
```

Collection and Stream API

- The collection represents the group of object representation also it is referred as single entity
- Stream is used to process from the collection
- IO Stream is different from Stream API (IO Stream talks about data or binary data buAPItream api the talks about stream of collection data)
- A Stream represents a sequence of elements from a source and supports various data processing operations. In other words, it provides an abstraction over an existing collection.
- //if you want to use stream

  Stream str = col.stream(); //java.util.Stream;

Creating a Stream from a Collection using stream() method. For example:

```
List<String> castList = List.of("Sam","Dean","Castiel","Crowley");
Stream<String> supernatural = castList.stream();
```

Collection and Stream API

Creating a Stream from an Array using stream() method. For example:

```
Integer[] array = {672, 340, 999};
Stream<Integer> stream = Arrays.stream(array);
```

Creating a Stream directly using of() method. For example:

Creating a Stream directly for Integers:

```
Stream<Integer> stream = Stream.of(672, 340, 999);
```

Creating a Stream directly for Objects of User-Defined Classes:

```
//Creating a Stream for objects of Class Employee
 Stream<Employee> empStream = Stream.of( new Employee("Tom",5699.5),
                                  new Employee("Jack",7629.2),new Employee("Jane",5289.8));
```

Collection and Stream API

```
class Employee{
        private String name;
        private Double salary;
        public Employee(String name, Double salary) {
                super();
                this.name = name;
                this.salary = salary;
        }
}
```

Collection and Stream API

Example1: filter on collection

```
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(10);
al.add(15);
al.add(20);
al.add(25);
al.add(30);
System.out.println(al);
List<Integer> al1 = al.stream().filter(i->i%2==0).collect(Collectors.toList());
System.out.println(al1);
```

Collection and Stream API

example 2: map on collection

```
ArrayList<Integer> mapList = new ArrayList<Integer>();
mapList.add(10);
mapList.add(15);
mapList.add(20);
mapList.add(25);
mapList.add(30);
System.out.println(mapList);
List<Integer> updatedMarks = mapList.stream().map(i->i+5).collect(Collectors.toList());
System.out.println(updatedMarks);
```

```
stream().filter(). map(). collect();
count();   sorted();
```

Collection and Stream API

example3: count on collectin

```
ArrayList<Integer> mapList = new ArrayList<Integer>();
mapList.add(40);
mapList.add(25);
mapList.add(30);
mapList.add(25);
mapList.add(35);
System.out.println(mapList);
long failedCount = mapList.stream().filter(i->i<35).count();
System.out.println(failedCount);
```

Collection and Stream API

example4: sorted on collection

```
ArrayList<Integer> mapList = new ArrayList<Integer>();
mapList.add(40);
mapList.add(25);
mapList.add(30);
mapList.add(25);
mapList.add(35);
System.out.println(mapList);
List<Integer> sortedList = mapList.stream().sorted().collect(Collector.toList());
System.out.println(sortedList);
```

Collection and Stream API

Comparator -->compare(obj1, obj2)
returns -ve if obj1 has to come before obj2
returns +ve if obj1 has to come after obj2
returns o if obj1 and obj2 are equal

(i1,i2)->(i1<i2)?1:(i1>i2)?-1:0
Comparable interface internally contains compareTo method
i1,i2 -->i1.compareTo(i2)

sorted(Comparator)==>customized sorting

Collection and Stream API

```
List<Integer> sortedList = mapList.stream().sorted((i1,i2)->i1.compareTo(i2)).collect(Collector.toList());
  System.out.println(sortedList);

   List<Integer> sortedList = mapList.stream().sorted((i1,i2)->i2.compareTo(i1)).collect(Collector.toList());
  System.out.println(sortedList);
```

(s1, s2) --> s1.compareTo(s2) ==>Natural Sorting order

(s1, s2) -->s2.compareTo(s1) ==>reverse natural sorting order

(s1,s2) --> s1.compareTo(s2) ==>reverse natural soroting order

Collection and Stream API

```
ArrayList<Integer> mapList = new ArrayList<Integer>();
  mapList.add(40);
  mapList.add(25);
  mapList.add(30);
  mapList.add(25);
  mapList.add(35);
  System.out.println(mapList);
  Integer min = mapList.stream().min((i1,i2)->-i1.compareTo(i2)).get();;
  System.out.println(min);

  Integer max = mapList.stream().max((i1,i2)->-i1.compareTo(i2)).get();;
  System.out.println(max);
```

Collection and Stream API

```
print iteration
  mapList == 10 items

  for(Integer i: i1)[
   System.out.println(i1);
  }

  i1.stream().forEach(function)
  i1.stream().forEach(System.out::println);
  i1.stream().forEach(i-> {
        System.out.println("The square of"+i+" is " +(i*i));
      });
```

```
//convert to integer array obect
 Integer[] i = i1.stream().toArray(Integer[]:new);
for(Integer i1:1){
  System.out.println(i1);
}

 //integer to stream
 Stream.of(i).forEach(System.out::println);

//Pipelining of streams
  Stream<Integer> intStream = Stream.of(10,2,7,5,6,5,8,11);
intStream.filter(n -> { System.out.println("Filtering Current Element: "+n); return n % 2 == 0; })
          .map(n -> { System.out.println("Mapping Current Element: "+n); return n * n * n; })
                              .sorted()
                              .forEach(n -> System.out.println(n));
```

Collection and Stream API

```
Stream<Integer> s = Stream.of(9,99,999,9999);
s.forEach(System.out:: println);

Integer[] i= {10,20,30,40,50};
Stream.of(i).forEach(System.out::println);
```

Pipelining of Streams

Now that we know about various operations of Streams and how to chain them together, let us know more about Stream Pipelining and Intermediate and Terminal operations.

Predefine Functional Interface

java.util.Function

predicates - boolean result

```
Predicate<String> checkLength = str -> str.length()>5
        System.out.println(checkLength.test("mallikarjuna")); // return true
        System.out.println(checkLength.test("tes"); //return false
```

consumer -no result

Pipelining of Streams

Consumer - modifies -no data

```
        class Person{
          private String name;
          setName(String name)
          getName();
}
 Person person = new Person();
 Consumer<Person> setName = t -> setName("Mallik");
        setName.accept(person);
        System.out.println(p.getName());
```

functions -input and output

```
Function<Integer,String> getInt= t-> t*10+"data multiplierd by 10";
System.out.println(getInt.apply(10));
```

supplier -no input only supply

```
Supplier<Double> rand = ()->Math.random();
System.out.println(rand.get());
```

functions -input and output

```
Function<Integer,String> getInt= t-> t*10+"data multiplierd by 10";
System.out.println(getInt.apply(10));
```

supplier -no input only supply

```
Supplier<Double> rand = ()->Math.random();
System.out.println(rand.get());
```

JDBC is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database.

- JDBC
  - - is the Java object version of Microsoft ODBC.
  - - defines Java class wrappers for SQL database access.
  - - can access almost any database service.
  - - JDBC drivers are available from almost any Java vendor.
  - java.sql.* package is now part of the Java core.
  - - compliance means the drivers support SQL92.
  - - must implement the java.sql.* classes.
  - - must be thread safe.

JDBC is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database.

- JDBC
  - is the Java object version of Microsoft ODBC.
  - defines Java class wrappers for SQL database access.
  - can access almost any database service.
  - JDBC drivers are available from almost any Java vendor.
  - java.sql.* package is now part of the Java core.
  - compliance means the drivers support SQL92.
  - must implement the java.sql.* classes.
  - must be thread safe.

Year	JDBC Version	JSR Specification	JDK Implementation
2017	JDBC 4.3	JSR 221	Java SE 9
2014	JDBC 4.2	JSR 221	Java SE 8
2011	JDBC 4.1	JSR 221	Java SE 7
2006	JDBC 4.0	JSR 221	Java SE 6
2001	JDBC 3.0	JSR 54	JDK 1.4
1999	JDBC 2.1		JDK 1.2?
1997	JDBC 1.2		JDK 1.1?

- JDBC 2.0
  - Support for data types, such as objects, arrays, and large objects (LOBs). This is handled through the standard java.sql package.
  - Support for standard features, such as result set enhancements and update batching. This is handled through standard objects, such as Connection,ResultSet, and PreparedStatement, under JDK 1.2.x and later.
  - Support for extended features, such as features of the JDBC 2.0 optional package, also known as the standard extension application programming interface (API), including data sources, connection pooling, and distributed transactions
- JDBC 3.0
  - Transaction Savepoints
  - Retrieval of Auto-Generated Keys
  - JDBC 3.0 LOB Interface Methods
  - Result Set Holdability

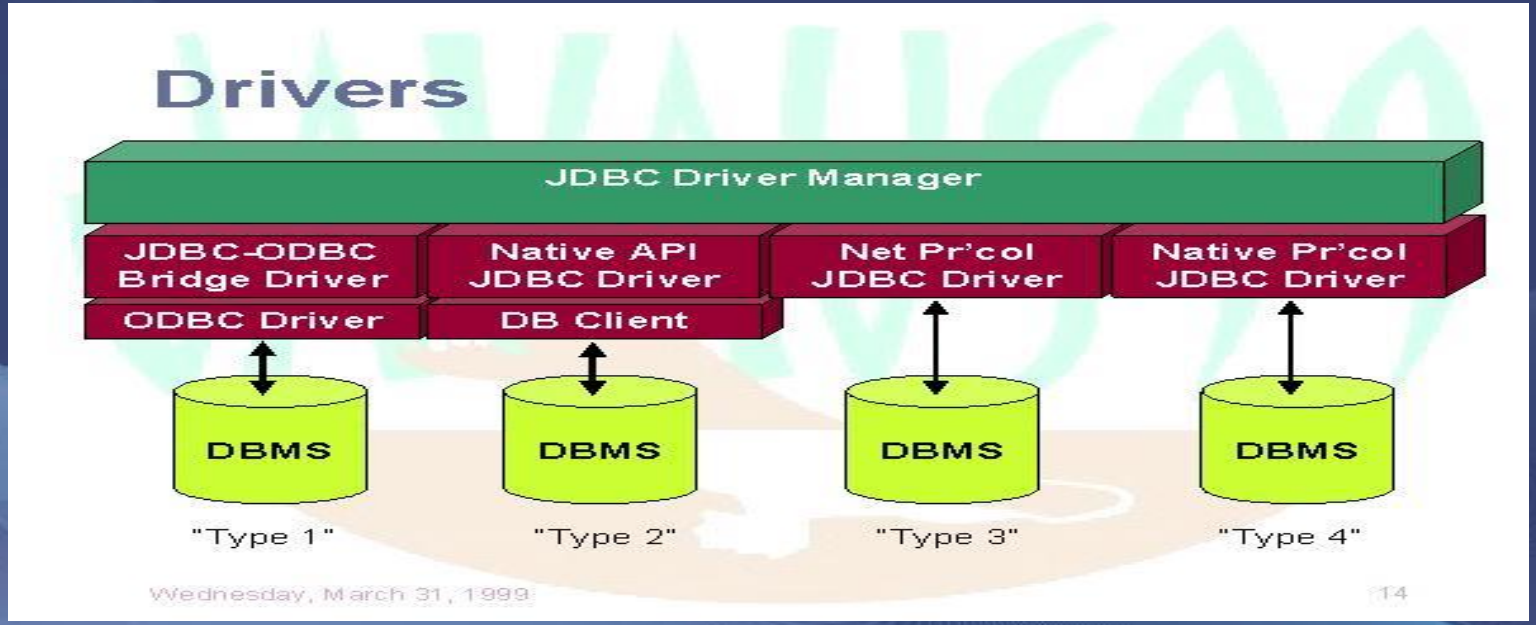
- JDBC 4.0
  - Wrapper Pattern Support
  - Enhanced Exception Hierarchy and SQLException
  - The RowId Data Type
  - LOB Creation
  - National Language Character Set Support
- JDBC 4.1
  - The ability to use a try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement
  - RowSet 1.1: The introduction of the RowSetFactory interface and the RowSetProvider class, which enable you to create all types of row sets supported by your JDBC driver.
- JDBC 4.2
  - Added support for REF CURSOR
  - Addition of the java.sql.DriverAction interface
  - Addition of the java.sql.SQLType interface
  - Addition of the java.sql.JDBCType Enum
  - Some JDBC API changes

- JDBC 4.3
  - Added support for Sharding
  - Addition of the java.sql.ConnectionBuilder interface
  - Addition of the java.sql.ShardigKey interface
  - Addition of the java.sql.ShardingKeyBuilder interface
  - Addition of the javax.sql.XAConnectionBuilder interface
  - Addition of the javax.sql.PooledConnectionBuilder interface

- Classes
  - locate DBMS drivers - DriverManagerClass
    - locates the driver for DB you specify
  - establish connections
    - commit, rollback and the DB isolation level
  - submit SQL statements
    - 2 extensions
      - Prepared statement - precompile and execute
      - Callable statement - stored procedures
  - process result set
    - manipulate the cursor and get back results

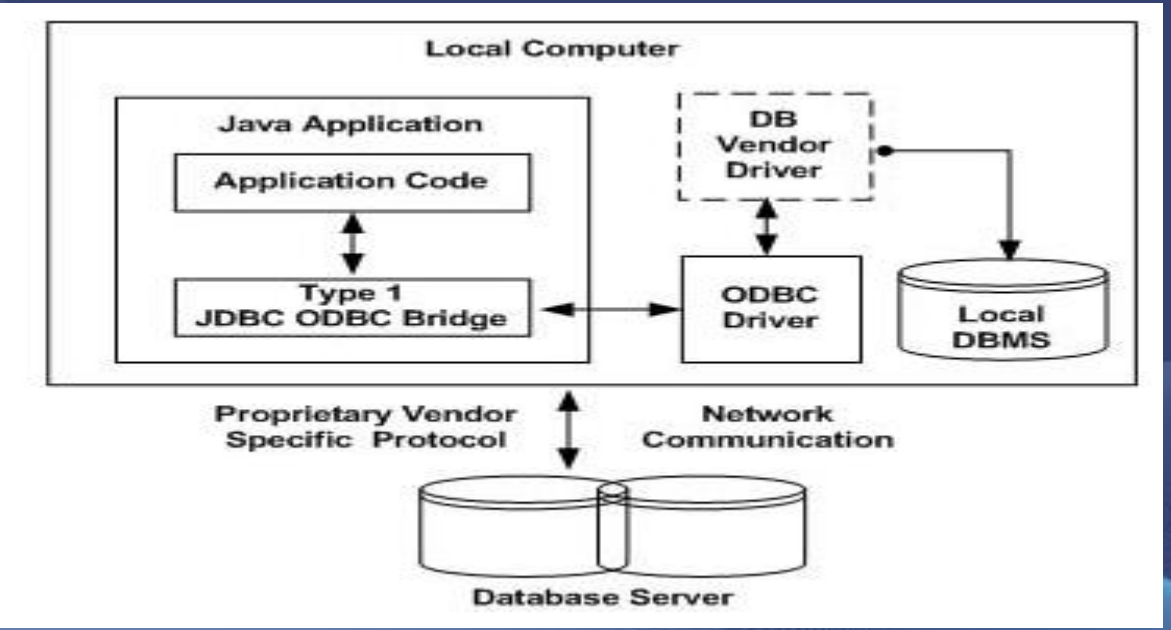
JDBC DRIVER

- JDBC drivers implement the defined interfaces in the JDBC API for interacting with your database server
- - Using JDBC drivers enable you to open database Connections and interact with them by sending SQL or database commands and then receiving results with Java



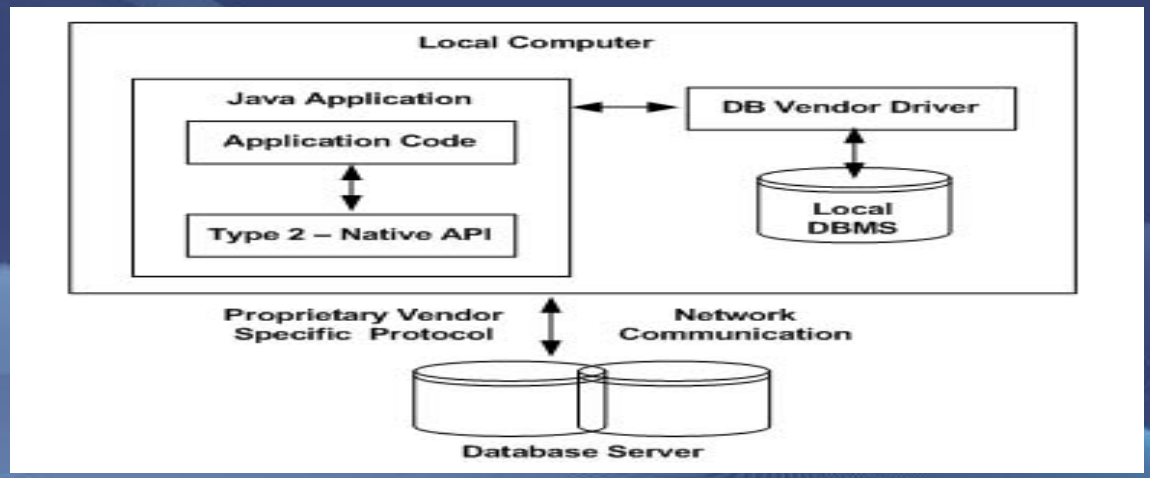
JDBC TYPE1 DRIVER

- This driver converts JDBC calls to ODBC calls through the JDBC-ODBC Bridge driver which in turn converts to database calls. Client requires ODBC libraries.
- a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.



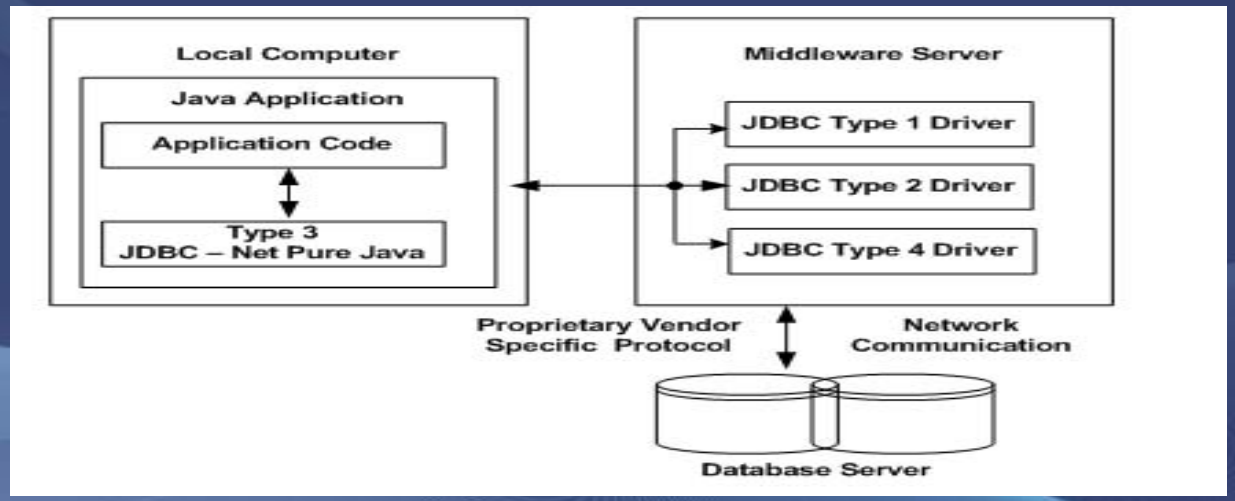
JDBC TYPE2 DRIVER

- This driver converts JDBC calls to database-specific native calls. The client requires database-specific libraries.
- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine
- E.g:- Oracle call Interface(OCI)



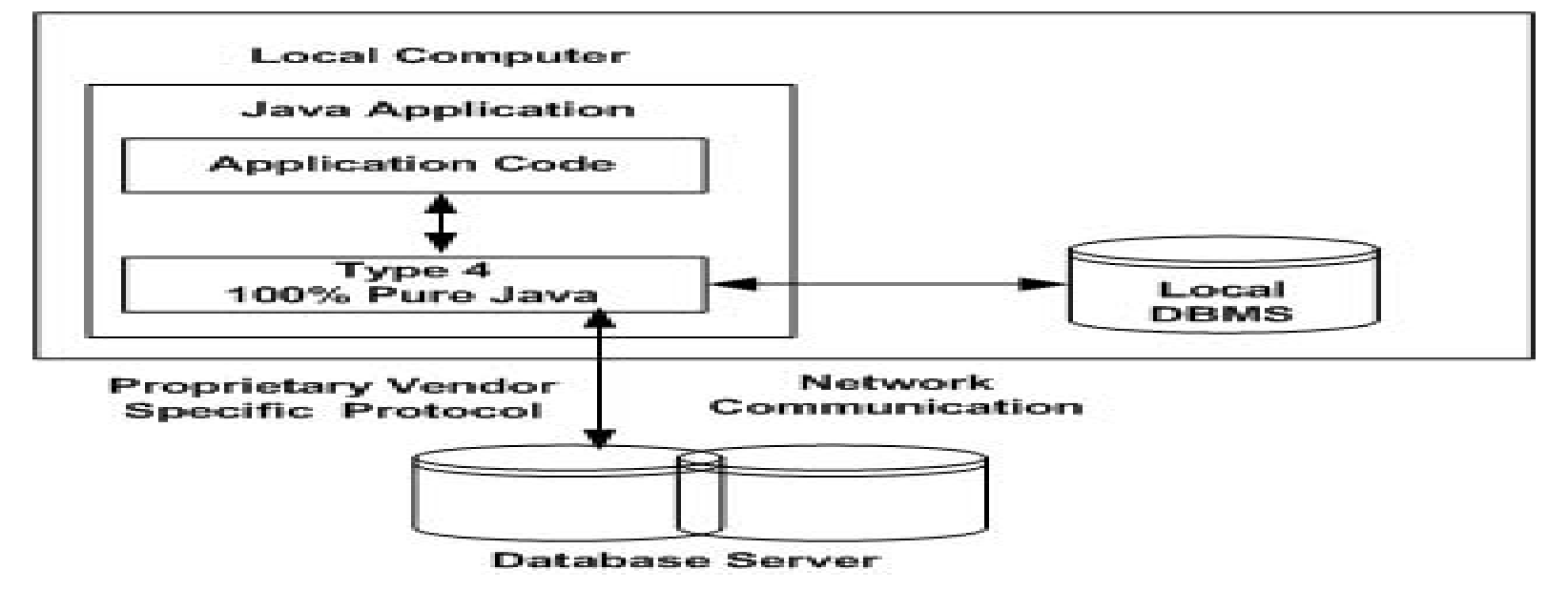
JDBC TYPE3 DRIVER

- This driver passes calls to a proxy server through network protocol which in turn converts to database calls and passes through the database-specific protocol. The client doesn't require any driver.
- In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with a middleware application server



JDBC TYPE4 DRIVER

- This driver directly calls the database. The client doesn't require any driver.
- In a Type 4 driver, a pure Java-based driver that communicates directly with the vendor's database through a socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.



BENCHMARKING JDBC DRIVER

- Type 4 - Accessing one type of database, such as Oracle, Sybase, or MySQL.
- Type 3 - Accessing Multiple types of database at the same time
- Type 2 - In case 3 or 4 not available on your machine
- Type 1 - Drivers are not considered for deployment, only for development and testing

TYPICAL STEPS

- Import the JDBC Package
- Set useful static class variables for driver, DBMS URI, credentials, etc.
- Register appropriate driver with the DriverManager class
- Set connection Properties (typically credentials)
- Invoke the Connection
- Create SQL statement in a String
- Invoke Statement on connection
- Execute SQL statement via appropriate method on Statement object.

Continued…

TYPICAL STEPS

- execute(...), executeQuery(...) for SELECT to receive ResultSet/s.
- executeUpdate(...) for INS/DEL/UPD or DDLs, receive integer value indicating row count.
- automatic commit, unless disabled, in which case an explicit COMMIT must be executed to save changes to the database.
- Check for Exceptions.
- Process ResultSet
- Close ResultSet and Statement
- Unless multiple transactions, then redefine and reuse
- Cycle for further operations
- Close Connection

- oracle.jdbc.driver (classes to support database access and updates in Oracle type formats)
- Load the Oracle JDBC driver

  DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

  or

  Class.forName ("oracle.jdbc.driver.OracleDriver");
- Connect to the database

  Connection conn = DriverManager.getConnection ("jdbc:oracle:thin:@learning", “root", “pwd");

- Code for creating a connection using JDBC with Oracle

```
try
{     // register the driver to connect to oracle
        DriverManager.registerDriver (new        oracle.jdbc.driver.OracleDriver());
        // db_con is a connection object that lets us connect to the
        // database with specified URL/userid/password
        Connection db_con = DriverManager.getConnection
        ("jdbc:oracle:thin:@llama.its.monash.edu.au:1521:CSE  5200A", "studentid", "password");
        System.out.println("Connection Success");
} catch(SQLException f ){
        // gives error if the above did not succeed.
        System.out.println("Connection Failed. SQL Exception");
```

- }

- Code to retrieve data from the Database using the previously created JDBC Connection object.

```
try
{       // statement object stmt created to execute queries using        //db_con
        Statement stmt = db_con.createStatement();

        // resultset is an object that holds the result of the query
        ResultSet rset = stmt.executeQuery ("SELECT * FROM      authors");

        // looping through the resultset object till end
        while (rset.next()){
        // getString gets the string data in col 1 of result
        // and we would have rset.getInt(2) if col 2 was an integer
        System.out.println (rset.getString(1));
        }
 } catch(SQLException f) {
        // error handling for above
        System.out.println("Cannot retrieve data SQL Exception          occured");
    }
```

- Statement selectStmt = dbConnection.createStatement();
- ResultSet result = selectStmt.executeQuery(selectText);
- insertText = "INSERT INTO addresses VALUES (1,'John','Smith')";
- Statement insertStmt = dbConnection.createStatement();
- int rowsInserted = insertStmt.executeUpdate(insertText);
- PreparedStatement
    - statement is pre-compiled and stored in a PreparedStatement object. Can use this to efficiently execute the statement multiple times
    - allows IN parameters within the query
    - referenced by number
    - setXXX () methods used
- setXXX();
  - pstmt.setString (1, "SCOTT");
- getXXX();
  - System.out.println(rset.getString (1));
- setNULL(); - don't use nulls

Executing a Query and Returning the ResultSet Interface

- ResultSet rset = stmt.executeQuery ("SELECT ename FROM emp");
- executeUpdate
  - insert, delete, update
- execute():
- while (rset.next())

  System.out.println (rset.getString(1));
- This is standard JDBC syntax. The next() method returns false when it reaches the end of the result set. The employee names are materialized as JavaStrings

Executing a Query and Returning the ResultSet Interface

- ResultSet rset = stmt.executeQuery ("SELECT ename FROM emp");
- executeUpdate
  - insert, delete, update
- execute():
- while (rset.next())

  System.out.println (rset.getString(1));
- This is standard JDBC syntax. The next() method returns false when it reaches the end of the result set. The employee names are materialized as JavaStrings
- You must explicitly close the ResultSet and Statement objects after you finish using them.
- rset.close();
- stmt.close();
- You must close your connection to the database once you finish your work.
- conn.close();

Executing a Query and Returning the ResultSet Interface

```
void dispResultSet (ResultSet rs) throws SQLException
{
 while (rs.next ())  {
        System.out.print ("" + rs.getInt (1))
        String firstName = rs.getString ("FIRST_NAME");
        System.out.print (rs.wasNull () ? “[no first name]” : firstName);
        System.out.print (rs.getString ("LAST_NAME"));
        System.out.println (rs.getDate (4));
 }
        rs.close ();
}
```

- getMessage(): returns the error message associated with the object that threw the exception
- printStackTrace(): prints this object name and its stacktrace to the specified print stream
- This example uses both getMessage() and printStackTrace() to return errors.

```
catch(SQLException e); {
        System.out.println("exception: " + e.getMessage());
          e.printStackTrace();
          while(e != null) {
                    System.out.println(e.getMessage());
                    e = e.getNextException();
          }
}
```

THANK YOU