Abstract: - The paper presents the facilities of Java programming language in Business Applications and emphasizes aspects like: connecting to databases, implementing business logic, performing different tasks, encapsulate objects and display results to the end user. Most applications have their main components structured in modules that represent business logic and Java programming language through classes, interfaces and services can implement them very easy. Because the business logic and the design are very important for the groups of developers, software architects and the end users, choosing Java as programming platform is a good option regarding the costs, splitting the tasks, enhance the source code and independency from system operating. Also, Java programming language permits to create custom classes, to build personal connection drives, to choose the way of building reports and to deploy applications on different types of servers.

Key-Words: - Business objects, Java objects integration, Java Virtual Machine (JVM), Custom driver connection and Java beans.

1 Introduction
Java programming language permits building business applications, in special Web applications, regardless of system operations and can access different types of data that can be structured in databases or semi-structured in XML files. Java is a programming language object oriented that offers many possibilities in the domain of accessing data, is very adaptive, is open-source, can be used with any type of operating system, it can be expand through the creation of new classes, new packages, new libraries by a large community of programmers, it extends the capacities and possibilities of Web servers and also is the most popular and robust programming language for building business applications. Java in business applications through specific drives can access data from databases that are managed by different RDBMS like MySQL, Microsoft SQL Server, Oracle, Microsoft Access, etc. and so these applications can be built for dedicated Web servers and the data becomes available for a large group of end users that are located in different places [1], [5].

Java makes possible to display dynamic content in Web pages through specialized instruments like JSP (JavaServer Pages), JSF (JavaServer Faces), servlets, etc. and by displaying data from databases. Another facility of Java is the possibility of accessing semi-structured data from XML files. Data structured in XML files are very popular in business applications because the management of those files is easy and the technology open-source is much appreciated in comparison with traditional RDBMS. A trend for the producers of RDBMS is to make available the possibilities of transforming various types of data in XML files that are more easily to manage, to store and used by the object-oriented programming languages [2], [4].

2 Java possibilities for connection at different types of data
Java programming language offers many possibilities to connect at different types of data through a built in driver called JDBC (Java Database Connectivity) which offers access to databases from different vendors and can manipulate the data from the tables through SQL statements which are arguments to methods in Java interfaces. Java provides database programmer’s features such as easy object to relational mapping, database independence, distributed computing, etc. The results from the database are returned as Java objects, and access problems get thrown as exceptions, so these objects can be used in many types of applications.

The most efficient way to customize applications is to build the own specific connection driver which can be used for accessing different types of data from various locations. My own solution is to build a custom Java class to connect a client at a server through sockets. Sockets represent a way to speak to other programs using standard Unix/Linux file descriptors. A file descriptor is an integer associated with an open file. That file can be a network connection, a pipe, a terminal, a real on-the-disk file, etc. When it is called a socket() system routine, it returns the socket descriptor, and the program communicate through it
using the specialized send() and recv() socket calls. There are two types of sockets. One type is stream sockets and the other type is datagram sockets, which may be referred to as SOCK_STREAM and SOCK_DGRAM. Datagram sockets are called connectionless sockets. Stream sockets are reliable two-way connected communication streams. They will also be error free [3], [5].

Stream sockets achieve high level of data transmission quality through a protocol called The Transmission Control Protocol, known as TCP. TCP makes sure that data arrives sequentially and error-free. Cool. Datagram sockets are called connectionless because if it is send a datagram, it may arrive and it may arrive out of order. If it arrives, the data within the packet will be error-free. Datagram sockets use IP (Internet Protocol) for routing, but they don’t use TCP, they use the User Datagram Protocol, or UDP. They are connectionless because it doesn’t have to maintain an open connection as it does stream sockets. A packet can be built, use an IP header on it with destination information, and send it out. No connection needed. They are generally used for packet-by-packet transfers of information [1], [3].

The most important feature in building a driver is the client-server architecture. The client connects to the server and the server is waiting for connections. The main differences are in the moment of the initialization not during the normal functioning. The sockets make possible communication in the network and contain an IP address, a protocol (TCP, UDP) and a port (number that permits separations of a several connections) [2]. The source code for the client that is part of the custom driver is:

```java
package con_cl_serv;
import java.io.*;
import java.lang.*;
import java.util.*;

class hel {
    byte[] hostent_intr;
}
class cntr_addr {
    byte[] sockaddr_intr; // connector’s address information
}
class prog_client {
    int PORT; // the port client will be connecting to
    int MAXDATASIZE = 100; // max number of bytes we can get at once
    int con_client(String cl_hostname, String serv_adr, String serv_port) {
        int sock_intr, numbytes;
        String buf[MAXDATASIZE];
        String[] hostent_intr, he;
        String[] sockaddr_intr, cntr_addr; // connector’s address information
        if (cl_hostname.compareTo("") == 0) {
            System.out.println("error usage: client hostname" + stderr);
            return 1;
        }
        if ((he=gethostbyname(serv_adr)) == NULL) { // get the host info
            System.out.println("error gethostbyname");
            return 1;
        }
        if ((sock_intr = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            System.out.println("error socket");
            return 1;
        }
        byte[] cntr_addr_in_family = AF_INET;
        // host byte order
        byte[] cntr_addr_in_port = htons(PORT);
        // short, network byte order
        String[] cntr_addr_in__addr = hel(hostent_intr);
        memset((cntr_addr_in_zero),'\0', 8);
        if(connect(sock_intr, sockaddr_intr,sizeof(sockaddr)) == -1) {
            System.out.println("error connect");
            return 1;
        }
        if ((numbytes=recv(sock_intr, buf, MAXDATASIZE-1, 0)) == -1) {
            System.out.println("error recv");
            return 1;
        }
        buf[numbytes] = "/0";
        System.out.println("Received: ",buf);
        close(sock_intr);
        return 0;
    }
}

The source code for the server that is part of the custom driver is:

```
class contr_addr {
    byte[] sockaddr_intr;
}

class sa {
    byte[] saction;
}

class prog_serv {
    void singlechild_handler(int s) {
        while(wait(NULL) > 0);
    }

    int con_serv() {
        int sockdb1, new_db1; // listen on sockdb1, new connection on new_db1
        String sockaddr_intr, addr_intr; // the address information
        String sockaddr_intr, contr_addr; // connector’s address information
        int s_size;
        String saction, sa;
        int ok=1;

        if ((sock_db1 = socket(AF_INET,
            SOCK_STREAM, 0)) == -1) {
            System.out.println("error socket");
            return 1;
        }

        if (setsockopt_db1(sock_db1,SOL_SOCKET,SO_REUSEADDR,ok,sizeof(int)) == -1) {
            System.out.println("error set socket opt");
            return 1;
        }

        byte[] addr_intr_in_family = AF_INET; // host byte order
        byte[] addr_intr_in_port = htons(PORT); // short, network byte order
        String addr_intr_in_addrs = INADDR_ANY; // automatically fill with the IP
        memset((addr_intr_in_zero), '\0', 8); // zero the rest of the addr

        if (bind(sockdb1, addr_intr,
            sizeof(sockaddr_intr)) == -1) {
            System.out.println("error bind");
            return 1;
        }

        if (listen(sockdb1, BACKLOG) == -1) {
            System.out.println("error listen");
            return 1;
        }

        String sa_handler = singlechild_handler(1); // reap all dead processes

        sigemptyset(sa);
        String sa_flags = SA_RESTART;

        if (sigaction(SIGCHLD, sa, NULL) == -1) {
            System.out.println("error sigaction");
            return 1;
        }

        while(1) { // main accept() loop
            int sin_size = sizeof(sockaddr_intr);

            if (bind(new_intr, accept(sockdb1,
                sockaddr_intr, addr_intr, sin_size)) == -1) {
                System.out.println("error accept");
                continue;
            }

            System.out.println("server: got connection from " +
                inet_ntoa(their_addr.sin_addr));

            if (!fork()) { // this is the child process
                close(sockfd); // child doesn’t need the listener

                if (send(new_intr, "Message first", 14, 0) == -1) {
                    System.out.println("error send");
                    close(new_intr);
                    return 1;
                }

                close(new_intr);
                return 0;
            }
        }
    }
}

The source code for the program talker - listener that is part of the custom driver is:

package con_talker_listener;

import java.io.*;
import java.lang.*;
import java.util.*;

class addr_intr {
    byte[] sockaddr_intr; // address information
}

class contr_addr {
    byte[] sockaddr_intr; // connector’s address information
}

class prog_listen {
int PORT; // the port users will be connecting to
int MAXBUFLEN = 100;

int con_listener(String cl_hostname, String serv_adr, String serv_port) {
    int sockdb1;
    int addr_len, numbytes;
    char buf[MAXBUFLEN];

    if ((sockdb1 = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        System.out.println("error socket");
        return 1;
    }

    byte[] addr_intr_in_family = AF_INET; // host byte order
    byte[] addr_intr_in_port = htons(PORT); // short, network byte order
    String addr_intr_in_addrs = INADDR_ANY; // automatically fill with the IP
    memset((addr_intr_in_zero), '\0', 8); // zero the rest of the addr

    if (bind(sockdb1, addr_intr, sizeof(sockaddr_intr)) == -1) {
        System.out.println("error bind");
        return 1;
    }

    addr_len = sizeof(sockaddr_intr);

    if ((numbytes = recvfrom(sockdb1, buf, MAXBUFLEN-1, 0, (sockaddr_intr), addr_len)) == -1) {
        System.out.println("error recvfrom");
        return 1;
    }

    System.out.println("got packet from " + addr_intr_in_addrs);
    System.out.println("packet is bytes long " + numbytes);

    buf[numbytes] = "/0";
    System.out.println("packet contains " + buf);
    close(sockdb1);
    return 0;
}

The source code for talker:

class hel {
    byte[] hostent_intr;
}

class cntr_addr {
    byte[] sockaddr_intr; // connector’s address information
}

class prog_talker {
    int PORT; // the port users will be connecting to

    int con_talk(String cl_hostname, String serv_adr, String serv_port) {
        int sock_intr, numbytes;

        if ((cl_hostname.compareTo("") == 0 && serv_adr.compareTo("") == 0 && serv_port.compareTo("") == 0) {
            System.out.println("error usage: talker hostname message" + stderr);
            return 1;
        }

        if ((he1 = gethostbyname(serv_adr)) == NULL) {
            System.out.println("error gethostbyname");
            return 1;
        }

        if ((sock_intr = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
            System.out.println("error socket");
            return 1;
        }

        byte[] cntr_addr_in_family = AF_INET; // host byte order
        byte[] cntr_addr_in_port = htons(PORT); // short, network byte order
        String[] cntr_addr_in_addr = hel1(hostent_intr);
        memset((cntr_addr_in_zero), '\0', 8);

        if ((numbytes = sendto(sock_intr, serv_adr, strlen(serv_adr), 0, sockaddr_intr, sizeof(sockaddr)) == -1) {
            System.out.println("error sendto");
            return 1;
        }

        System.out.println("sent " + numbytes + " bytes to " + cntr_addr_in_addr);
        close(sock_intr);
        return 0;
    }
}

In the code above in the server program the socket is specified, the protocol is sock_stream, the method bind associates socket listen with the variable serv_addr, the method bzero fills with zero the variable serv_addr. The server program doesn’t need to initialize the explicit IP address, only the client needs to do that and the method
listen makes the server specification. The most important methods for the server program are: socket(), bind(), listen(), accept(), read and write. In the client program the correspondent methods are socket(), bind(), connect(), read and write [2], [6].

The advantage of using a custom driver is that the programmer has the full control over the design and implementation of the applications, with little changes can access every type of data and also this driver can be improved and used by a large number of software developers because is a Java class and so is open-source. In the source code of this driver could be implemented various methods for managing data in databases (select, insert, update, delete, create, etc.) or from XML files.

JDBC accomplishes its tasks through a set of Java interfaces, each implemented differently by individual vendors. The set of classes that implement the JDBC interfaces for a particular database engine is called a JDBC driver. The idea of JDBC is to hide the specifics of each database and lets the programmers to concentrate on their applications. A database query for any database engine it requires connection to the database, issue a SELECT statement, and process the result set. Here is an example for a simple SELECT application from the Imaginary JDBC Driver for m1SQL. This application is a single class that gets all of the rows from a table in an m1SQL database. First, it connects to the database by getting a database connection under a user id, from the JDBC DriverManager class. It uses that database connection to create a Statement object that performs the SELECT query. A ResultSet object then provides the application with the key and val fields from the test1 table. The source code for using the JDBC driver:

```java
import java.sql.*;

public class cls1 {
    public static void main(String args[]) {
        String url = "jdbc:m1sql://test_db.com/sql1";
        Connection con = null;

        try {
            String driver = "com.test_domain.sql.m1sql.M1sqlDriver";
            Class.forName(driver).newInstance();
            }
        catch( Exception e ) {
            System.out.println("Failed to load m1SQL driver.");
            return;
        }
        try {
            con = DriverManager.getConnection(url, "user_name", "");
            Statement sel1 = con.createStatement();
            ResultSet result1 = sel1.executeQuery("select t_f1, t_v1 from test1");
            System.out.println("the results:");
            while(result1.next()) { // process results
                int key1;
                String val1;
                key1 = result1.getInt(1);
                if( result1.wasNull() ) { key1 = -1; }
                val1 = result1.getString(2);
                if( result1.wasNull() ) { val1 = null; }
                System.out.println("key1 = " + key1);
                System.out.println("val1 = " + val1);
            }
            System.out.println("key1 = "+ key1);
            System.out.println("val1 = "+ val1);
        }
        catch( Exception e ) {
            e.printStackTrace();
        }
        finally {
            if( con != null ) {
                try { con.close(); } catch( Exception e ) { e.printStackTrace(); }
            }
        }
    }
}
```

The one phase when it is hard to achieve portability is the first step of connecting, because it must be specify a driver. An application uses JDBC as an interface through which it passes all its database requests. When is written a Java database applet or application, the only driver-specific information JDBC requires is the database URL. The JDBC Connection process is the most difficult part of JDBC to get right. The API itself is fairly straightforward, but many things hide right beneath the surface. The new JDBC Standard Extension will cover a simplified way of making database connections that avoids many of these problems. If the problems appear just making a connection, check if they match any of the following:

- Connection fails with the message "Class not found". This message usually results from not having the JDBC driver in the CLASSPATH. To solve this problem it must be enter .zip and .jar files explicitly into a CLASSPATH.
- Connection fails with the message "Driver not found". In this case it must be register the JDBC driver with the DriverManager class. When using the Class.forName() method of registering a JDBC driver, it can be encounter an inconsistency between the JDBC specification and some JVM implementions. So it can be use the Class.forName().newInstance() method as a workaround [1], [5].

Using the database URL and whatever properties the JDBC driver requires (generally a user ID and password), the application will first request a java.sql.Connection implementation from the
DriverManager. The DriverManager in turn will search through all of the known java.sql.DriverManager implementations for the one that connects with the URL that has been provided. If it exhausts all the implementations without finding a match, it throws an exception back to the application. Once a Driver recognizes the URL, it creates a database connection using the properties that have been specified. It then provides the DriverManager with a java.sql.Connection implementation representing that database connection. The DriverManager then passes that Connection object back to the application. JDBC requires a Driver class to register itself with the DriverManager when it is instantiated. The act of instantiating a Driver class thus enters it in the DriverManager's list. Instantiating the driver, however, is only one of several ways to register a driver:

- Explicitly call new to load your driver's implementation of Driver. In other words, the programmer hardcode the loading of a Driver implementation in the application. His alternative is the least desirable since it requires a rewrite and recompile if the database or database driver changes.
- Use the jdbc.drivers property. The DriverManager will load all classes listed in this property automatically. This alternative works well for applications with a command-line interface, but might not be so useful in GUI applications and applets. While environment variables do work for GUI applications, it cannot be done in Java applets [4], [6].
- Load the class using Class.forName("DriverImplementationClass") newInstance();

This complex expression is a tool for dynamically creating an instance of a class when some variable are representing the class name. Because a JDBC driver is required to register itself whenever its static initializer is called, this expression has the net effect of registering the driver. For now, Class.forName("classname") is supposed to be sufficient, but some Java virtual machines do not actually call the static initializer until an instance of a class is created. As a result, newInstance() should be called to guarantee that the static initializer is run for all virtual machines.

3 Conclusion

For different types of applications the programmer may need to access various types of data available in RDBMS's (relational database management system) that seems to have their own ways of representing specific content. The Java programming language uses the JDBC (a SQL-level API) for accessing data and allows programmers to implement business logic through embedded SQL statements that are arguments to methods in JDBC interfaces. In the databases applications built on Java platform, JDBC offers database connectivity and makes the translation of relational data into objects [3]. Even if the databases engines changes, the logic of Java applications that uses JDBC does not need to change, because the library classes maps its objects to databases entities in such a way that the logic of applications does not even know whether or not their objects are being stored in databases. JDBC translates between the entities of the database and the objects of the Java application. The results from the databases are returned as Java objects, and access problems get thrown as exceptions, so the applications are very flexible and permits customizations in every moment of their existence [4]. Using a custom driver for connection (own driver) has many advantages like accessing different types of data, create the own objects, manage various structures of data including XML files and because this driver is built with Java programming language makes it open-source, easy to customize by other software developers and can be improved by implementing different methods and classes [1]. Using Java for business applications makes possible to split work for designers, software developers by using business objects encapsulated in classes and so is easy to manage, to modify, to improve and to create new modules for different type of end users. Java makes possible to upgrade old business applications and to build new ones that are more stable and much easier to use by many users and is also platform free.

References:


URI: http://java.sun.com
URI: www.j2ee.me/docs/books/tutorial/jdbc/TOC.html
URI: www.easysoft.com/products/data_access/odbc_jdbc_gateway/
URI: http://javaworld.com/javaworld/
URI:http://www.packtpub.com/service-oriented-java-business-integration/
URI: http://www.sun.com/software/javaenterprisesyste/
URI: http://www.manageability.org/