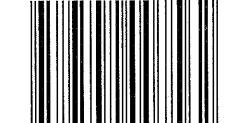


import org.hibernate.HibernateException;
import org.hibernate.Session;

ISBN 978-954-20-0756-2



9 789542 007562 >

Цена: 4,30 лв.

User getUser

new Connection

String password){

SessionFactory();

Session session = session

Transaction tx = null;

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

public class User {

private static SessionFactory



Технически университет – Варна

Катедра "Компютърни науки и технологии"

Христо Божидаров Ненов

Технически университет – Варна
Катедра “Компютърни науки и технологии”

Христо Божидаров Ненов

**МРЕЖОВО ПРОГРАМИРАНЕ С
JAVA**

Справа:

4.4.2 Създаване на обект от тип Statement.....	74
4.4.3 Обработка на резултата. ResultSet.....	75
4.4.4 Затваряне на връзката с базата.....	75

Предговор

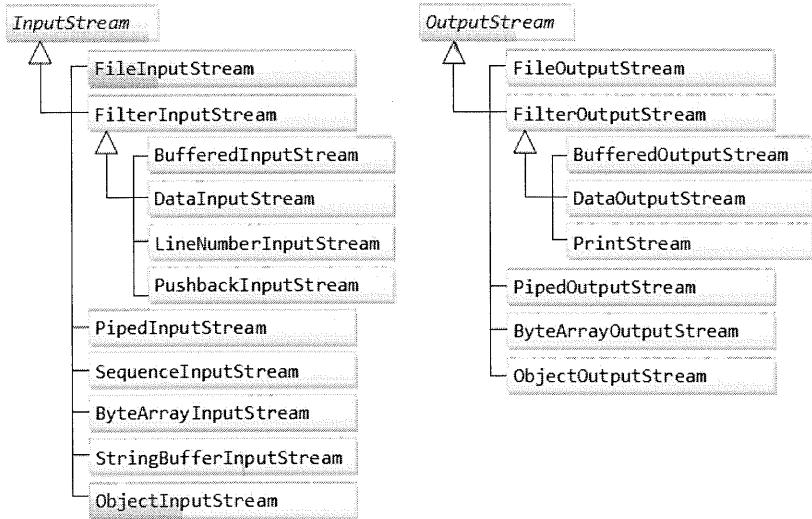
Ръководството за лабораторни упражнения по лисциплината

1. Входно-изходни потоци в Java.

1.1. Потоци за вход и изход в Java.

Ролята на голяма част от мрежовите програми е да извършват обикновен процес на вход и изход (I/O): преместване на байтове (информация) от една система към друга. Процесът на четене на информацията, изпратена от сървър, не е много различен от процеса за четене на файл, или изпращането на текст към клиент не се различава много от запис във файл. Въпреки това, I/O в Java е организиран различно от типичните програмни езици като C, C++, Fortran и др. В Java за I/O се използва концепцията на потоците: входният поток чете (приема) информация; изходният поток записва (изпраща) информация. Потокът е абстракция, която или

- Символно-ориентирани потоци – интерпретират информацията като символи;
- Буферирани потоци (Buffered Streams) – оптимизиране на работата чрез редуциране на обръщенията към системните API;
- Потоци за форматирани данни (Scanning and Formatting) – позволяват четене и запис на форматиран текст;
- Стандартни потоци (I/O from the Command Line) – работи със стандартен вход и изход;
- Информационни потоци (Data Streams) – имат възможност да интерпретират поредица от байтове като примитивни данни типове или String;



Фиг.1.1 Йерархия на класовете за работа с байтово-ориентирани потоци.

Освен в пакета `java.io` има дефинирани различни специфични класове за работа с потоци и в други пакети. Например `java.util.zip` дефинира следните класове, които работят с компресиране и декомпресиране на информация:

- `CheckedInputStream`
- `CheckedOutputStream`
- `DeflaterOutputStream`
- `GZIPInputStream`
- `GZIPOutputStream`
- `InflaterInputStream`
- `ZipInputStream`
- `ZipOutputStream`

В `java.util.jar` са дефинирани два класа за работа с JAR архиви:

- `JarInputStream`
- `JarOutputStream`

Java Cryptography Extension (JCE) добавя два класа за криптиране и декриптиране на информация:

- `CipherInputStream`
- `CipherOutputStream`

Използване на байтово-ориентиран поток.

Пример 1: Копиране на информация между файлове чрез байтово-ориентиран поток.

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}

```

Затварянето на потока след като вече не е необходим е от изключително важно значение*. В примера „CopyBytes“ този процес е поставен в блока „finally“, което гарантира тяхното

* От версия Java 1.7 е дефиниран интерфейсът „AutoCloseable“, който дава възможност на програмиста да декларира I/O операциите без да се грижи за затварянето на потоците. По-подробно надолу в текста.

затваряне независимо от това дали успешно или не ще се изпълни предходния код. Тази практика спомага за избягването на проблема с изтичане на памет. В примера, за по-лесно онагледяване на идеята, е използвано твърдо задаване на имената на файловете. Зададени точно по този начин те трябва да бъдат поставени на определено място – в главната директория на проекта или с други думи в директорията, от която ще се стартира програмата ако се използва: `java „filename“ program`“. Възможен проблем при изпълнението на програмата е тя да не може да отвори единия или другия файл, които използва. В този случай началната стойност, която имат (`null`), няма да се промени. Поради тази причина се извършва

които се базират на два байта Unicode код единици, а не на еднобайтови символи.

Символните потоци са инструмент за организиране на входно/изходни операции с данни, които се интерпретират като символи. Те използват Unicode и следователно могат да бъдат интернационализирани. В някои случаи символно-ориентираните потоци са по-ефикасни от байтово-ориентиранияте. Първата версия на Java (Java 1.0) не е включвала символни потоци, всички са били байтово-ориентирани. Символните потоци се появяват във версия Java 1.1, което налага някои от класовете и методите на байтово-ориентиранияте потоци да бъдат забранени.

Използване на символно-ориентиран поток

Пример 2: Копиране на информация между файлове чрез символно-ориентиран поток.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader fin = null;
        FileWriter fout = null;

        try {
            inputStream = new FileReader("input.txt");
        }
```

които съществуват, са по отношение на интерпретиране и трансформиране на данните, които пропадат през потока. Например програма, която работи със .zip архиви, трябва да може да осъществи връзка към файла върху файловата система и прочитане на двоичната информация (*FileInputStream*), коректна интерпретация на архивния файл (*ZipInputStream*), извлечане на същинската информация (напр. *BufferedInputStream*) и евентуалното ѝ асоцииране с различните данни типове в Java (*DataInputStream*). Това се постига чрез изграждане на филтри от каскадно вложени класове.

Пример 3: Четене от файл чрез използване на вложени класове.

```
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

В пример 3 е представено влагането на класове като на създаването на обект от тип *DataInputStream* се подава като аргумент обект от тип *BufferedInputStream*, на който пък е подаден обект от тип *FileInputStream*, който на свой ред е приел аргумент от тип *File*.

1.4 Автоматично затваряне на потоци. Използване на „try“ с ресурс. Интерфейсът “AutoCloseable”.

Във версия Java 1.7 бе представена концепцията за автоматично затваряне (освобождаване) на ресурси. Под ресурс в Java се разбира обект, който трябва да се затвори или освободи след приключването на работата с него. За да може един обект да бъде

```
System.out.println(str);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

1.5 Разбиване на символни потоци на отделни елементи.

Основните методи, които се използват при работа със символно-ориентирани низове са *read()* и *readLine()*. Това води до следният проблем: при работа с този тип низове в много от случаите се изиска интерпретирането на отделните елементи, присъстващи в един ред от текстовия файл, вместо целия ред да бъде представен като един цял обект от тип *String*. За решението на този проблем в Java има няколко подхода:

От друга страна, въпреки че се счита за вече остаряла технология, StringTokenizer не бива да бъде подценяван заради лесния начин за използване.

Разделяне на символен низ на поднизове

Пример3: Демонстрация на работа със StringTokenizer, String.split и Scanner

```
import java.io.*;
import java.util.Scanner;
import java.util.StringTokenizer;

public class SubstringsDemo {

    public static String delimiter = ",";
    public static File file = new File("substrings.txt");

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
        FileReader(file));
        System.out.print("Original text from file: ");
        System.out.println(br.readLine());
        SubstringsDemo sub = new SubstringsDemo();
        sub.readWithStringTokenizer();
        sub.StringSplit();
        readWithScanner();
    }

    public void readWithStringTokenizer() {
        try (BufferedReader br = new BufferedReader(new
        FileReader(file))) {
            String line;
            while ((line = br.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(line,
delimiter);
                System.out.println("Result from StringTokenizer:");
                while (st.hasMoreElements())
                    System.out.println(st.nextElement());
            }
            System.out.println();
        } catch (java.io.IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
    }

    public void StringSplit() {
        try (BufferedReader br = new BufferedReader( new
FileReader(file))) {
            String line;
            System.out.println("Result from String.split:");
            while ((line = br.readLine()) != null) {
                String[] split = line.split(delimiter);
                for (int i = 0; i < split.length; i++) {
                    System.out.println(split[i]);
                }
                System.out.println();
            }
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }

    public static void readWithScanner() {
        Scanner sc = null;
        try {
            String line;
            sc = new Scanner(file);
            sc.useDelimiter(delimiter);
            System.out.println("Result from Scanner:");
            while (sc.hasNext()) {
                System.out.println(sc.next());
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Изход:

```
Original text from file: one,,two,tree,4,5
Result from StringTokenizer:
one
two
tree
4
5
Result from String.split:
```

```
one  
two  
tree  
4  
5  
Result from Scanner:  
one  
  
two  
tree  
4  
5
```

Независимо от полученият резултат в конкретната задача, `String.split()` и `Scanner` имат възможност за използване на сложни регулярни изрази, които биха дали коректен резултат за доста по-сложни символни низове от демонстрирания.

1.6 Сериализация.

Сериализацията в Java присъства като технология от версия Java 1.1 и е една от важните възможности, които предоставя езикът. Процесът на сериализация представлява конвертиране на обект в байтова последователност, която може да се изпрати чрез поток по мрежа към файл или към база от данни за по-късно използване. Обратният процес, десериализация, е конвертирането на поток от „обекти“ в конкретни Java обекти, използвани в текущата програма.

За да може един обект да бъде сериализиран, той

```
private String firstName;  
private transient String middleName;  
  
private String lastName;  
  
public Person(String firstName, String middleName, String  
              lastName) {  
    this.firstName = firstName;  
    this.middleName = middleName;  
    this.lastName = lastName;  
}  
  
public void setFirstName(String firstName) {  
    this.firstName = firstName;  
}  
  
public void setMiddleName(String middleName) {  
    this.middleName = middleName;  
}  
  
public void setLastName(String lastName) {  
    this.lastName = lastName;  
}  
  
public String getMiddleName() {  
    return middleName;  
}  
  
public String getLastname() {  
    return lastName;  
}
```

SerializationDemo.java

```
import java.io.*;  
  
public class SerializationDemo {  
    public static void main(String[] args) {  
        String filename = "file.ser";
```

5. Какъв е резултата при сериализация на обект, деклариран като *transient*?

6. Във файла „employee.dat“ има разположени два реда с информация, организирана в следната структура:

1 | James | Gosling | 25 | 0

2 | Harry | Hacker | 10 | 0

2. Многозадачност в Java.

2.1 Същност.

Използването на паралелна обработка е неделима част от съвременния ИТ свят. Все по-завишените изисквания на потребителите, както и стандартите за производителност, които се налагат от само себе си, правят немислимо съвременното

При първия подход тежестта на разпределение на изпълнението пада върху операционната система. Принципът на действие е на всички нишки през определен интервал от време да се дава възможност за изпълнение. В Java като стратегия за изпълнение на нишки е избран втория подход (cooperative). При този подход се предоставя на програмиста да определи последователността на изпълнение на нишките. Java предоставя директен достъп до

- *Thread()*
- *Thread(String<name>)*

В първият случай операционната система автоматично ще зададе име на нишката, във втория това става експлицитно от програмиста. Например:

```
Thread firstThread = new Thread();
Thread secondThread = new Thread("namedThread");
System.out.println(firstThread.getName());
System.out.println(secondThread.getName());
```

ще генерира следния изход:

```
Thread-0
namedThread
```

2.2.2 Имплементиране на интерфейса *Runnable*.

Създаването на нишка посредством имплементиране на интерфейса *Runnable* е много близко до предходния вариант. Първо се създава клас, който имплементира интерфейса. След това се инстанцира обект от този клас като се обвива в обект от тип *Thread*. Това става като се създаде обект от тип *Thread* и на конструктора му като аргумент се подаде новосъздадения обект, имплементиращ *Runnable*. Тук отново се използват предимно следните два конструктора:

- *Thread(Runnable<object>)*

```
RunnableHelloCount threadDemo =
    new RunnableHelloCount();

public RunnableHelloCount() {
    thread1 = new Thread(this);
    thread2 = new Thread(this);
    thread1.start();
    thread2.start();
}

public void run() {
    int pause;
    for (int i = 0; i < 10; i++) {
        try {
            System.out.println(Thread.currentThread().getName()
                + " being executed.");
            pause = (int) (Math.random() * 3000);
            Thread.sleep(pause);
        } catch (InterruptedException interruptEx) {
            System.out.println(interruptEx);
        }
    }
}
```

2.3 Многонишков сървър

Има едно основно и важно ограничение, свързано с всички

Използваният от по-дълго време (и все още широко използван – Apache Tomcat Server) е подходът на многонишковия сървър. Той има няколко важни предимства:

- предлага "чисто" изпълнение чрез отделяне на задачата за създаване на нишки от тази за обработка на връзките;
- устойчив, тъй като при проблем с една от връзките, това няма да се отрази на обработката на другите връзки.

Основната техника включва процес на два етапа:

1. главната нишка (диспечер) назначава индивидуална нишка за всяка постъпила клиентска заявка;
2. назначената нишка поема изцяло комуникацията между клиента и сървъра.

Създаването на нишка е бавен процес в Java. Това налага използването на различни техники, които имат за цел ефективното боравене с нишки. Една от тези техники е така наречения пул от нишки.

Пример 1: Реализация на echo сървър с използване на нишки за комуникация с множество клиенти.

MultiEchoServer.java

```
public class MultiEchoServer {

    private static ServerSocket serverSocket;
    private static final int PORT = 1300;

    public static void main( String[] args ) throws IOException
    {
        try {
            serverSocket = new ServerSocket( PORT );
        } catch( IOException ex ) {
            System.out.println( "\nUnable to set up port!" );
            System.exit( 1 );
        }

        do {
            Socket clientSocket = serverSocket.accept();
            System.out.println("\nNew client accepted!\n" );
            ClientHandler handler = new ClientHandler(
                clientSocket );
        }
    }
}
```

```
        handler.start();
    } while( true );
}
}
```

MultiEchoClient.java

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class MultiEchoClient {
    private static InetAddress host;
    private static final int PORT = 1300;

    public static void main( String[] args ) {
        try {
            host = InetAddress.getLocalHost();
        } catch( UnknownHostException uex ) {
            System.out.println( "\nHost ID not found!\n" );
            System.exit( 1 );
        }
        ClientMessenger.sendMessages( host, PORT );
    }
}
```

ClientHandler.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

class ClientHandler extends Thread {

    private Socket socket;
    private Scanner input;
    private PrintWriter output;

    public ClientHandler( Socket socket ) {
        this.socket = socket;
    }

    try {
        input = new Scanner( socket.getInputStream() );
        output = new PrintWriter(
            socket.getOutputStream(), true );
    } catch( IOException ex ) {
        ex.printStackTrace();
    }
}
```

```

@Override
public void run() {
    String received = "";
    do {
        //accept message
        received = input.nextLine();
        //return message
        output.println( "ECHO: " + received );
    } while( !received.equals( "QUIT" ) );
    try {
        if ( socket != null ) {
            System.out.println( "Closing down connection" );
            socket.close();
        }
    } catch( IOException ex ) {
        System.out.println( "Unable to disconnect!" );
    }
}
}

```

ClientMessenger.java

```

import java.io.IOException;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.Socket;
import java.util.Scanner;

class ClientMessenger {

    public static void sendMessages(
        InetAddress address, int port ) {
        Socket socket = null;
        try {
            socket = new Socket( address, port );

```

```

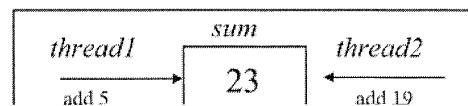
('QUIT' to exit): " );
message = userEntity.nextLine();
output.println( message );
response = input.nextLine();
System.out.println( "SERVER> " +
response );
} while ( !message.equals( "QUIT" ) );

input.close();
userEntity.close();
} catch (IOException e) {
e.printStackTrace();
} finally {
try {
System.out.println( "Closing connection..." );
socket.close();
} catch( IOException ex ) {
System.out.println( "Unable to disconnect!" );
System.exit( 1 );
}
}
}
}

```

2.4 Взаимно блокиране. Locks и Deadlock.

Многонишковите програми могат да предизвикат някои проблеми, дължащи се на необходимостта за координиране дейностите на различните нишки, които се изпълняват в рамките на програмата. На Фиг. 2.1 е показан пример за това, какво може да се обърка в дадена ситуация. Тук нишките *thread1* и *thread2* актуализират общ ресурс – “*sum*”.



необходимото в това копие и след това да запише новата стойност обратно в „*sum*“. Крайната стойност от двете първоначални операции трябва да бъде 47 (23+5+19). Ако обаче се случи и двата процеса на четене да се изпълнят преди една от актуализациите, ще се получи презаписване на резултата, който няма да е коректен: в

2.5 Синхронизиране на нишки

Мониторинг на обект в Java позволяващ неговото заключване се постига чрез поставяне на ключовата дума *synchronized* пред декларация на метод или блок от код, който прави актуализирането. Например:

DA-2020-0000000000000000000000000000

DA FORM 2020 DA FORM 2020

```

        System.out.println( "Producer: the new level is " +
newLevel );
pause = (int) (Math.random() * 5000);
sleep( pause );
} catch( InterruptedException ex ) {
    System.out.println( ex );
}
} while( true );
}
}

```

Resource.java

```

class Resource {

    private int numResources;
    private final int MAX = 5;

    public Resource( int startLevel ) {
        this.numResources = startLevel;
    }

    public int getLevel() {
        return numResources;
    }

    public synchronized int addOne() {
        try {
            while( numResources >= MAX ) {
                wait();
            }

            numResources++;

            notifyAll();
        } catch( InterruptedException ex ) {
            System.out.println( ex );
        }

        return numResources;
    }

    public synchronized int takeOne() {
        try {
            while( numResources == 0 ) {
                wait();
            }

            numResources--;
        }
    }
}

```

```

        notify();
    } catch( InterruptedException ex ) {
        System.out.println( ex );
    }

    return numResources;
}
}

```

ResourceServer.java

```

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class ResourceServer {

    private static ServerSocket server;
    private static final int PORT = 1300;

    public static void main( String[] args ) throws IOException
    {

        try {
            server = new ServerSocket( PORT );
        } catch( IOException ioEx ) {
            System.out.println( "\nUnable to set up port!" );
            System.exit( 1 );
        }

        Resource item = new Resource( 1 );
        Producer producer = new Producer( item );
        producer.start();

        do {
            Socket client = server.accept();
            System.out.println( "\nNew client accepted.\n" );

            Consumer consumer = new Consumer( client, item );
            consumer.start();
        } while( true );
    }
}

```

ResourceClient.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

public class ResourceClient {
    private static InetAddress host;
    private static final int PORT = 1300;

    public static void main( String[] args ) {
        try {
            host = InetAddress.getLocalHost();
        } catch( UnknownHostException uex ) {
            System.out.println( "\nHost ID not
found!\n" );
            System.exit( 1 );
        }

        sendMessages( host, PORT );
    }

    public static void sendMessages(
        InetAddress address, int port ) {
        Socket socket = null;

        try {
            socket = new Socket( address, port );
            Scanner input = new Scanner(
                socket.getInputStream() );

            PrintWriter output = new PrintWriter(
                socket.getOutputStream(), true );
            Scanner userEntity =
                new Scanner( System.in );
            String message = "", response = "";

            do {
                System.out.println( "Enter 1 for
resource or 0 to quit: " );
                message = userEntity.nextLine();

                if ( message.equals( "1" ) ) {
                    output.println( "Get" );
                    response = input.nextLine();
                    System.out.println( response );
                }
            } while ( !message.equals( "0" ) );
        } catch( IOException e ) {
            e.printStackTrace();
        } finally {
            try {
                System.out.println( "Closing
connection..." );
                socket.close();
            } catch( IOException ex ) {
                System.out.println( "Unable to
disconnect!" );
                System.exit( 1 );
            }
        }
    }
}
```

```
System.out.println("*Invalid message ***" );
}
} while ( !message.equals( "0" ) );

input.close();
userEntity.close();
} catch( IOException e ) {
e.printStackTrace();
} finally {
try {
    System.out.println( "Closing
connection..." );
    socket.close();
} catch( IOException ex ) {
    System.out.println( "Unable to
disconnect!" );
    System.exit( 1 );
}
}
}
```

2.6 Неблокиращи сървъри.

J2SE 1.4 въвежда нов програмен интерфейс за входно-изходни операции, обозначен с абревиатурата NIO. Този интерфейс е имплементиран от пакета `java.nio` и неговите подпакети, от които с най-съществено значение е `java.nio.channels`. Вместо традиционните входно-изходни потоци в Java, NIO предоставя възможност за реализация на концепцията на каналите. За разлика от потоците, които са байтово ориентирани, каналите са блоково ориентирани. Това означава, че информацията може да бъде пренасяна под формата на големи блокове от данни вместо на индивидуални байтове, което от своя страна води до драстично увеличение на скоростта на предаване. Всеки канал е асоцииран със собствен буфер, който осигурява съхранение на данните, които

системно ниво. Това позволява избягването на междинни буфери и води до изключително увеличение на скоростта на пренос на данните.

Вместо да заделя индивидуална нишка за всеки клиент, NIO използва мултиплексиране (обработване на множество връзки едновременно от единственный обект). Това се базира на използването на селектори (единствения обект), които наблюдават както създаването на новите връзки, така и трансфера на данни по вече създадените такива. Всеки канал се регистрира посредством селектор за събитие, към което проявява интерес. Режимите, в които могат да се използват каналите, са два: блокиращ и неблокиращ. Използването на селектори за следене на събития означава, че вместо да се задава индивидуална нишка за всяка връзка, се използва една (или повече ако поискаме), която следи няколко канала едновременно. Това спомага за избягването на проблеми като лимити от операционната система, взаимни блокировки или погрешно достъпване на общ ресурс, както и проблеми, характерни за подхода „една нишка за връзка“.

2.6.1 Реализация на NIO.

Каналите, които се асоциират със *Socket* и *ServerSocket*, се наричат *SocketChannels* и *ServerSocketChannel*. Техните класове са част от пакета *java.nio.channels*. По подразбиране сокетите, асоциирани с такива канали, работят в блокиращ режим, но могат да бъдат конфигурирани и за неблокиращ като при извикването на метода *configureBlocking()* подадем като аргумент *false*. Това е метод на класа на канала, затова трябва да бъде извикан преди асоциирането му със сокет. След като е зададен режима, сокет може да се генерира с извикване на методът *socket()*.

Друг клас, който участва в NIO е класът *Selector*, който се намира също в пакета *java.nio.channels*. Обектът от този клас има ролята да следи за създаването на връзки и за трансфера на данните. Всеки канал (без значение дали *SocketChannel* или *ServerSocketChannel*) трябва да се регистрира със *Selector* за типа събитие, от което се интересува. Това става чрез извикването на

метода *register()*. Има четири статични константи от класа *SelectionKey* (*java.nio.channels*), служещи за определяне на типа на събитието, за което може да се следи:

- *SelectionKey.OP_ACCEPT*;
- *SelectionKey.OP_CONNECT*;
- *SelectionKey.OP_READ*;
- *SelectionKey.OP_WRITE*.

Тези константи са променливи от тип *int* побитово шаблонизирани, което позволява да се приложи логическо „ИЛИ“ (OR), за да бъде добавен и втори аргумент.

Кодът за създаване на обект от клас *Selector* има следната специфика – обектът е създаден не чрез конструктор а чрез извикване на статичен метод *open()*, което създава специфичен за платформата (операционната система) под-клас, който е скрит за програмиста.

```
selector = Selector.open();
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
.....
.....
.....
socketChannel.register(selector, SelectionKey.OP_READ );
```

Последната стъпка, която трябва да се направи, е да се зададе характеристиката на буфера (обект от клас *Buffer* (*java.nio*)). Класът *Buffer* е абстрактен клас, поради което всъщност се създава обект измежду някой от седемте му класа наследници:

- *ByteBuffer*;
- *CharBuffer*;
- *IntBuffer*;
- *LongBuffer*;
- *ShortBuffer*;
- *FloatBuffer*;
- *DoubleBuffer*.

С изключение на първия клас, всички останали са типово специфицирани. Класът *ByteBuffer* обаче поддържа четене и запис на останалите шест, което го прави най-широко използван поради

```
new InetSocketAddress( PORT );
serverSocket.bind( netAddress );

selector = Selector.open();
serverSocketChannel.register( selector );
```

```
Socket socket;  
  
socketChannel = serverSocketChannel.accept();  
socketChannel.configureBlocking( false );  
socket = socketChannel.socket();  
System.out.println( "Connection on " + socket + "." );  
socketChannel.register( selector, SelectionKey.OP_READ );
```

```
if ( socket != null ) {  
    socket.close();  
}  
} catch( IOException ioEx ) {  
    System.out.println( "*** Unable to close socket!  
***" );  
}
```

3. Мрежови потоци.Сокети. Класът iNet.

3.1 Класът iNet.

За работа в мрежова среда Java програмния език притежава редица конвенционални инструменти, както и такива, специфични за самия език. Един от тези инструменти е класът *InetAddress*, който е част от пакета *java.net*. Ролята на този клас е да работи с Интернет адреси, като има способността да ги интерпретира и като имена на хостове, и като IP адреси. Статичният метод *getByName()*, който класът притежава, използва DNS (Domain Name System) системата, за да върне Интернет адреса на специфичен хост. Резултатът, който се връща от метода, е обект от тип *InetAddress*. Тъй като използването на този метод може да доведе до изключението *UnknownHostException* при неразпознаване името на хоста, то трябва да се използват някой от двата стандартни метода за прихващане на изключения в Java (за предпочитане в този случай е да се използва *catch* блок).

Пример 1: *IPFinder.java*

```
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Scanner;

public class IPFinder {

    public static void main( String[] args ) {
        String host;
        Scanner is = new Scanner( System.in );
        InetAddress address;
        System.out.println( "Enter host name: " );
        host = is.next();
        try {
            address = InetAddress.getByName( host );
            System.out.format("\nHostname:%s",
                address.getHostName() );
            System.out.format( "\nIP address: %s",
                address.getHostAddress() );
            System.out.format( "\nCanonical Hostname: %s",
                address.getCanonicalHostName() );
        } catch ( UnknownHostException e ) {
            System.out.format("\nCould not find host%s", host );
        }
    }
}
```

```
        is.close();
    }
}
```

В примера по-горе е показан резултатът от работата на методите *getByName()*, *getHostName()*, *getHostAddress()*, *getCanonicalHostName()*, с които се взема съответно: името на хоста, IP адрес на хоста и пълното домейн име на този хост.

Изход:

```
Enter host name:
www.java.com

Hostname: www.java.com
IP address: 23.205.244.210
Canonical Hostname: www.java.com
a23-205-244-210.deploy.static.akamaitechologies.com
```

Понякога е необходимо да се получи информация за IP адреса на локалната машина. Това може да стане чрез метода *getLocalHost()*:

```
System.out.format( "\nLocalHost: %s", InetAddress.getLocalHost() );
```

3.2 Използване на сокети.

Различни процеси (програми) могат да комуникират една с друга дори и през множество от мрежи, които ги разделят. Технологията, която позволява това, са „сокетите“. Както вече бе разгледано в глава 1, сокета е елемента, който дефинира крайните две комуникационни точки. Java имплементира и двата вида сокети – **TCP/IP** и **Datagram** (UDP). Много често комуникацията между два процеса има „клиент-сървър“ характер. Стъпките за осъществяването на една такава комуникация на базата на изброените два типа сокета са много близки и почти идентични.

3.2.1 TCP/IP сокети.

Комуникационна връзка, създадена през TCP / IP сокети, е адресно ориентирана, затворена, „надеждна“ връзка. Това означава, че връзката между сървър и клиент остава отворена през цялата продължителност на диалога между двете страни и се нарушава само (при нормални обстоятелства) когато единия участник в диалога официално прекрати обмена на информация (чрез съгласуван протокол).

1. Създаване на *ServerSocket* обект.

Конструктора на класа *ServerSocket* изиска номер на порт (1024-65535) като аргумент. Например:

```
ServerSocket serverSocket = new ServerSocket(1234);
```

В този случай сървърът ще чака и слуша на порт 1234 за постъпване на заявка за комуникация.

2. Поставяне на сървъра в състояние на „чакане“.

Сървърът чака неопределено време за получаване на клиентска заявка. Това се осъществява чрез извикването на метода *accept()* на класа *ServerSocket*, който връща обект от *Socket* при

Аналогично, за получаване на изход може да се използва обект *PrintWriter* като обвивка на *OutputStream* обект, върнат от метода *getOutputStream()*. Например:

```
PrintWriter output = new PrintWriter(link.getOutputStream(),true);
```

Добавянето на втори аргумент в конструктора на *PrintWriter* – „*true*“ задава задължително почистване на буфера при всяко извикване на *println()* (което обикновено е желателно).

4. Изпращане и получаване на данни.

След като има вече настроени *Scanner* и *PrintWriter* обекти, получаването и изпращането на данни става изключително лесно. За получаване на данни ще бъде използван методът *nextLine()*, а за изпращане – *println()*. Например:

```
String input = input.nextLine();
output.println("Awaiting data...");
```

5. Затваряне на връзката след край на комуникацията.

При настъпване на край на комуникация следва задължителното затваряне на потоците. Това се осъществява

маркиране ако настъпят няколко изключения. По-добра практика изисква използването на два отделни блока: един за работа с порта, по който ще се комуникира и един за самата комуникация.

Добрата практика изиска и затварянето на сокета да е във "finally" блока, тъй като той ще се изпълни винаги, което гарантира, че сокетът ще бъде затворен при всякакви обстоятелства.

TCPEchoServer.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class TCPEchoServer {
    private static ServerSocket socket;
```

```
String message = input.nextLine(); // стъпка 4

while( !message.equals( "*CLOSE*" ) ) {
    System.out.println( "\nMessage received..." );
    numMessages++;
    output.println( "Message " + numMessages + ":" + message );
    message = input.nextLine();
}

output.println( "Messages received: " + numMessages ); // стъпка 4
} catch (IOException e) {
    e.printStackTrace();
} finally {
    System.out.println("\nClose connection...");
    input.close(); // стъпка 5
    try {
        link.close();
    } catch (IOException e) {
        System.out.println( "\nUnable to 
```

```

        System.exit( 1 );
    }

    accessServer();
}

private static void accessServer() {
    Socket link = null; //стъпка 1
    Scanner input = null;
    Scanner userEntry = null;
    try {
        link = new Socket( host, PORT ); //стъпка 1
        input =
new Scanner(link.getInputStream());//стъпка 2
        PrintWriter output =
new PrintWriter( link.getOutputStream(),
true );
        userEntry = new Scanner( System.in );
        String message, response;
        do {
            System.out.println( "Enter message: " );
            message = userEntry.nextLine();
            output.println( message ); // стъпка 3
            response = input.nextLine(); // стъпка 3
            System.out.println( "SERVER: " +
response );
        } while( !message.equals( "*CLOSE*" ) );

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        System.out.println( "Closing connection..." );
        input.close(); // стъпка 4
        userEntry.close();
        try {
            link.close();
        } catch (IOException e) {
            System.out.println( "Unable to
disconnect..." );
            System.exit( 1 );
        }
    }
}
}

```

3.2.2 Datagram (UDP) Сокети.

За разлика от TCP/IP, дейтаграм сокетите не са ориентирани към свързаност с конкретен хост или адрес.

1. Създаване на datagram сокет обект.

Както и при създаването на *ServerSocket*, процесът се състои от подаване на номер на порт като аргумент на конструктора на класа *DatagramSocket*.

Например:

```
DatagramSocket datagramSocket = new DatagramSocket(1234);
```

2. Създаване на буфер за приемане на datagram пакети.

Буферът за приемане на пакетите реално представлява масив от байтове, в който ще се съхрани входната информация. Например:

```
byte[] buffer = new byte[256];
```

3. Създаване на *DatagramPacket* обект за приемане на входящите datagram пакети.

Конструкторът на този обект изисква два аргумента:

- създавания в предходната стъпка масив от байтове (byte[] buffer);
- размера на същия масив.

Например:

```
DatagramPacket inPacket =
new DatagramPacket(buffer, buffer.length);
```

4. Приемане на входящите datagram пакети.

Приемането на пакетите се осъществява посредством извикването на методът *receive()* на *DatagramSocket* обекта, като за аргумент ще послужи вече създаденият *DatagramPacket* обект.

Например:

```
datagramSocket.receive(inPacket);
```

5. Извлечане на адреса и порта на изпращащият от получението пакет.

За извлечане на информацията относно адреса и порта на изпращащия процес се използват методите `getAddress()` и `getPort()` на обекта от тип `DatagramPacket`. Например:

```
InetAddress clientAddress = inPacket.getAddress();
int clientPort = inPacket.getPort();
```

6. Извлечане на данните от буфера.

За удобство при работа, данните се изтеглят като низ чрез използването на пренатоварената (overloaded) форма на `String` конструктора, който приема три аргумента:

- масив от байтове;
- началната позиция в рамките на масива (0);
- брой байтове (пълен размер на буфер).

Например:

```
String message = new String(inPacket.getData(), 0,
inPacket.getLength());
```

7. Създаване на пакет за отговор.

Създаването на `DatagramPacket` обект става посредством

8. Изпращане на отговора.

Процесът на изпращане се осъществява посредством извикването на метода `send()` на `DatagramSocket` обекта, като за аргумент служи създаденият `DatagramPacket` обект.

```
datagramSocket.send(outPacket);
```

Стъпки от 4 до 8 могат да бъдат изпълнени неограничен брой пъти (в рамките на един цикъл). При нормални обстоятелства, работата на сървъра най-вероятно няма да бъде спряна. Въпреки това, ако възникне изключение свързаният `DatagramSocket` трябва да бъде затворен, както е показано в стъпка 9 по-долу.

9. Затваряне на `DatagramSocket`.

Затварянето се осъществява посредством метода `close()` на `DatagramSocket` обекта.

Например:

```
datagramSocket.close();
```

За сравнение на двата подхода за комуникация (TCP и UDP), показаният по-горе пример тук е представен, реализирайки UDP комуникация. Между различните подходи има две основни разлики:

- `IOException` е заменен от `SocketException`;
- няма проверка за коректното генериране от метода

```

try {
    datagramSocket = new DatagramSocket( PORT );
} catch (SocketException e) {
    System.out.println( "Unable to open
                        port...\n" );
    System.exit( 1 );
}
handleClient();
}

private static void handleClient() {
    String messageIn, messageOut;
    int numMessages = 0;
    InetAddress clientAddress = null;
    int clientPort;
    try {
        do {
            buffer = new byte[256];
            inPacket =
                new DatagramPacket( buffer, buffer.length );
            datagramSocket.receive( inPacket );
            clientAddress = inPacket.getAddress();
            clientPort = inPacket.getPort();
            messageIn =
                new String( inPacket.getData(), 0,
                           inPacket.getLength() );
            System.out.println( "Message received: " );
            numMessages++;
            messageOut = "Message " + numMessages + ":" +
                         messageIn;
            outPacket =
                new DatagramPacket( messageOut.getBytes(),
                                   messageOut.length(), clientAddress,
                                   clientPort );
            datagramSocket.send( outPacket );
        } while( true );
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        System.out.println( "\nClosing connection..." );
        datagramSocket.close();
    }
}

```

TCPEchoClient.java

```

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Scanner;

public class UDPEchoClient {

    private static InetAddress host;
    private static final int PORT = 1230;
    private static DatagramSocket datagramSocket;
    private static DatagramPacket inPacket, outPacket;
    private static byte[] buffer;

    public static void main( String[] args ) {
        try {
            host = InetAddress.getLocalHost();
        } catch (UnknownHostException e) {
            System.out.println( "Host ID not found..." );
            System.exit( 1 );
        }
        accessServer();
    }

    private static void accessServer() {
        Scanner userEntry = new Scanner( System.in );
        try {
            datagramSocket = new DatagramSocket();
            String message = "", response = "";

            do {
                System.out.println( "Enter message: " );
                message = userEntry.nextLine();

                if ( !message.equals( "***CLOSE***" ) ) {
                    outPacket = new
                        DatagramPacket( message.getBytes(),
                                       message.length(), host, PORT );
                    datagramSocket.send( outPacket );

                    buffer = new byte[256];
                    inPacket = new DatagramPacket( buffer, buffer.length );
                    datagramSocket.receive( inPacket );
                    response = new String( inPacket.getData(), 0,
                                          inPacket.getLength() );
                    System.out.println( "SERVER: " + response );
                }
            } while( true );
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```

        } while( !message.equals( "***CLOSE***" ) );
    } catch ( IOException e) {
        e.printStackTrace();
    } finally {
        System.out.println( "Closing
connection...\n" );
        datagramSocket.close();
        userEntry.close();
    }
}
}

```

3.3 Използване на графичен интерфейс.

След като бяха разгледани основите на сокет програмирането в Java, следващата стъпка, която ще разгледаме, е добавяне на графичен потребителски интерфейс към програмите – нещо, с което е свикнал съвременния потребител. С цел вниманието в примерите да е насочено към интерфейса на програмите, а не към някакъв специфичен програмен фрагмент, са избрани примери, които дават достъп до стандартни услуги посредством „добре познати“ портове.

Пример 3:

Следващият пример, който ще бъде разгледан, предоставя информация за часа и датата чрез използване на стандартния *Daytime* протокол и порт 13 за комуникация. Посредством графичния интерфейс на програмата потребителя има възможност да посочи името на сървър, от който да бъде получена информацията.

GetRemoteTime.java

```

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;
import javax.swing.JButton;
import javax.swing.JFrame;

```

```

import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class GetRemoteTime extends JFrame implements ActionListener
{

    private static final long serialVersionUID = 1L;

    private JTextField hostInput;
    private JTextArea display;
    private JButton timeButton;
    private JButton exitButton;
    private JPanel buttonPanel;
    private static Socket socket = null;

    public static void main( String[] args ) {
        GetRemoteTime frame = new GetRemoteTime();
        frame.setSize( 400, 300 );
        frame.setVisible( true );
        frame.addWindowListener( new WindowAdapter() {

            public void windowClosing( WindowEvent e ) {
                if ( socket != null ) {
                    try {
                        socket.close();
                    } catch( IOException ex ) {
                        System.out.println( "\nUnable to close
link.\n" );
                        System.exit( 1 );
                    }
                }
                System.exit( 0 );
            }
        });
    }

    public GetRemoteTime() {
        hostInput = new JTextField( 20 );
        add( hostInput, BorderLayout.NORTH );

        display = new JTextArea( 10, 15 );
        display.setWrapStyleWord( true );
        display.setLineWrap( true );
        add(
            new JScrollPane( display ), BorderLayout.CENTER );
        buttonPanel = new JPanel();

```

```

timeButton = new JButton( "Get date and time" );
timeButton.addActionListener( this );
buttonPanel.add( timeButton );
exitButton = new JButton( "Exit" );

exitButton.addActionListener( this );
buttonPanel.add( exitButton );
add( buttonPanel, BorderLayout.SOUTH );
}

@Override
public void actionPerformed( ActionEvent e ) {
    if ( e.getSource() == exitButton ) {
        System.exit( 0 );
    }
    String theTime;
    String host = hostInput.getText();
    final int DAYTIME_PORT = 1300;

    try {
        socket = new Socket( host, DAYTIME_PORT );
        Scanner input =
new Scanner( socket.getInputStream() );
        theTime = input.nextLine();
        display.append( "The date/time at " + host +
" is " + theTime + "\n" );
        hostInput.setText( "" );
        input.close();
    } catch ( UnknownHostException uhEx ) {
        display.append( "No such host!\n" );
        hostInput.setText("");
    } catch ( IOException ioEx ) {
        display.append( ioEx.toString() + "\n" );
    } finally {
        try {
            if ( socket != null ) {
                socket.close();
            }
        } catch ( IOException e1 ) {
            System.out.println( "Unable to disconnect!" );
            System.exit( 1 );
        }
    }
}
}

```

С цел да се демонстрира работата на един „Daytime“ сървър, а и поради факта, че в днешно време е все по-трудно да се намерят такива, като допълнение ще бъде разгледан и пример на такъв.

DayTimeServer.java

```

import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;

public class DayTimeServer {

    public static void main( String[] args ) {
        ServerSocket server;
        final int DAYTIME_PORT = 1300;
        Socket socket;

        try {
            server = new ServerSocket( DAYTIME_PORT );
            do {
                socket = server.accept();
                PrintWriter output =
new PrintWriter( socket.getOutputStream(), true );
                Date date = new Date();
                output.println( date );
                socket.close();
            } while( true );
        } catch ( IOException e ) {
            System.out.println( e );
        }
    }
}

```

Въпроси и задачи:

1. Разгледайте и анализирайте примерите от упражнението.
2. Каква е разликата в имплементацията между използването на класове за работа TCP и UDP пакети?
3. Какво е knock-knock протокол? Реализирайте такъв за множество клиенти!
4. Разгледайте интерфейсът *Executor*. Изпълнете гореописаните примери с пул от нишки!

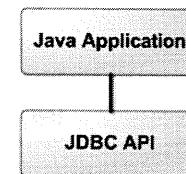
4. JDBC. Работа с бази от данни.

4.1 Същност.

Изключително голямото разнообразие на релационен тип бази от данни (най-широко използваният вид в наши дни) изправя програмистите пред един основен проблем – как да се приложи универсален подход, който да работи за коя да е Система за Управление на Бази от Данни (СУБД), съответно за нейния програмен интерфейс (API). Можем да се досетим, че вътрешния формат на една СУБД от Oracle ще е коренно различен от този на Microsoft Access, както и че те пък от своя страна нямат нищо общо с MySQL. Технологията, която предоставя Java по този въпрос и която решава напълно този проблем, се нарича JDBC.

JDBC – Java Database Connectivity, с индустриален стандарт

JDBC използва мениджър на драйверите и специфични за конкретна база от данни драйвери, за да осигури прозрачна свързаност към разнородни бази от данни. Мениджърът на JDBC драйверите гарантира, че се използва правилният драйвер за достъп до всеки източник на данни. Той е в състояние да поддържа едновременно множество драйвери, свързани с множество разнородни бази от данни. Архитектурата на JDBC е показана на фигура 4.1:



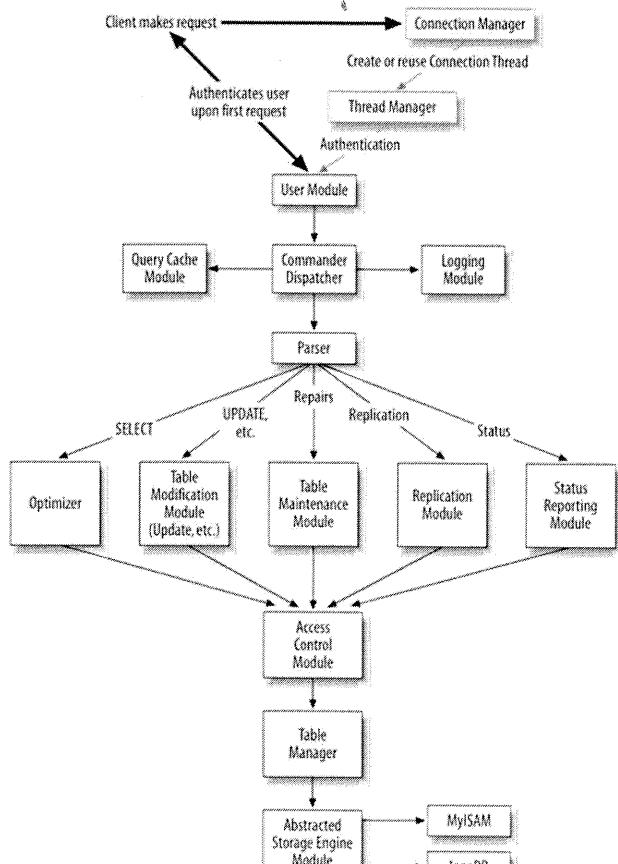
- Driver - осигурява комуникациите със сървъра на базата от данни. Програмистът няма прям достъп до това. Неговият достъп е до DriverManager обекти, които от своя страна управляват обекти от тип Driver. Скриване на детайлите по самата комуникация и осигуряване на ниво на абстракция са другите неща, които предоставя Driver;
- Connection - предоставя всички методи за комуникация с базата от данни. Обектът представя комуникационния контекст, т.е. цялата комуникация с база от данни е само чрез този обект;
- Statement: Обекти от този интерфейс се използват за подаване на SQL заявки към базата от данни. Някои производни на него интерфейси приемат и допълнителни параметри за изпълнение на съхранени процедури;
- ResultSet: Тези обекти съхраняват в себе си данните, извлечени от базата от данни, след като се изпълни SQL заявката с помощта на Statement обект. ResultSet действа като итератор, за да осигури достъп до данните, които сълържа:

4.3.1 MySQL.

MySQL е една от най-популярните Open Source SQL системи за управление на бази от данни, която се поддържа от Oracle Corporation. MySQL е придобил такава популярност поради различни причини:

- реализиран е под open-source лиценз и е напълно безплатен;
- поддържа големия набор от функционалности и инструменти на останалите скъпи и мощнни пакети за съхранение на данни;
- използва стандартна форма на SQL;
- работи върху много операционни системи и поддържа голям брой езици за програмиране;
- работи много бързо дори с голям обем данни;
- поддържа големи масиви от данни, до 50 милиона реда или повече в таблица. Големината на файловете му за таблица по подразбиране е 4GB, но може да бъде увеличена (ако операционната система позволява това) до теоретичните 8 милиона терабайта (TB).
- подлежи на модификации от страна на програмиста. Open-source

Архитектура на MySQL и работните процеси са показани на фигура 4.2.



След като процесите по инициализацията приключат, инициализиращият модул предава управлението на контролера за връзките (Connection Manager), който започва да слуша за заявки към базата на определения за това порт. Когато клиент се свърже към сървър, услугата на базата от данни „Connection Manager“ изпълнява поредица от процеси за комуникация на ниско ниво и предава на свой ред управлението на „Thread Manager“. Той създава нишка за обслужване на получената заявка или извика такава от пула с нишки, ако има свободна. Първото нещо, което прави обаче, е да извика User Authentication модула, за да провери характеристиките на потребителя, извършващ заявката до базата. Ако достъпа е одобрен, нишката предава данните към „Command Dispatcher“ и процесите по по-нататъчното обслужване на заявката продължават.

4.3.2 Даннови типове в MySQL.

Правилното определяне на типа и размера на полетата в таблица е важно за цялостната оптимизация на една база от данни. Тези видове полета (или колони) съответстват на различни типове данни, които ще се съхраняват в тях. MySQL използва много различни типове данни, които могат да се разделят най-общо в три категории: числови, дата и час, и символна.

1. Числови типове данни

MySQL използва стандартен ANSI SQL числов тип данни, аналогичен с другите СУБД:

- **INT** – стандартна променлива от целочислен тип (integer), която може да е със или без знак. Ако е със знак, то диапазона и ще е

- **SMALLINT** - стандартна променлива от целочислен тип (integer) за малки числа, която може да е със или без знак. Ако е със знак, то диапазона и ще е от -32768 до 32767. Ако е без знак, то диапазона става от 0 до 65535;
- **MEDIUMINT** - стандартна променлива от целочислен тип (integer) за средно големи числа, която може да е със или без знак. Ако е със знак, то диапазона и ще е от -8388608 до 8388607. Ако е без знак, то диапазона става от 0 до 16777215;
- **BIGINT** - стандартна променлива от целочислен тип (integer) за големи числа, която може да е със или без знак. Ако е със знак, то диапазона и ще е от -9223372036854775808 до 9223372036854775807. Ако е без знак, то диапазона става от 0 до 18446744073709551615;
- **FLOAT(M,D)** – за числа с плаваща запетая, който не може да бъде без знак, където M е дължината на числото (включително знаците след десетичната запетая), а D е брой знаци след десетичната запетая;
- **DOUBLE(M,D)** - за двойна точност при числа с плаваща запетая, който не може да бъде без знак, където M е дължината на числото (включително знаците след десетичната запетая), а D е брой знаци след десетичната запетая;
- **DECIMAL(M,D)** – непакетиран тип с плаваща запетая, който не може да бъде без знак. Всяко число отговаря на един байт. M е дължината на числото (включително знаците след десетичната запетая), а D е брой знаци след десетичната запетая. NUMERIC е синоним на DECIMAL.

2. Тип за дата и време

Типовете в тази категория са:

- **DATE** – дата във формат YYYY-MM-DD, между 1000-01-01 и 9999-12-31. December 30th, 1973 ще бъде съхранен като 1973-12-30;

- **DATETIME** – дата и час комбинация във формат YYYY-MM-DD HH:MM:SS, между 1000-01-01 00:00:00 и 9999-12-31 23:59:59. 15:30 December 30th, 1973 ще бъде съхранен като 1973-12-30 15:30:00;
- **TIMESTAMP** – timestamp между January 1, 1970 и 2037. Формат YYYYMMDDHHMMSS;
- **TIME** – време във формат HH:MM:SS;
- **YEAR(M)** – година в дву- (от 1970 до 2069) или четирицифрен формат (от 1901 до 2155). По подразбиране форматът е четирицифрен.

3. Символни низове.

Символните низове са данновия тип, който предимно се използва в MySQL:

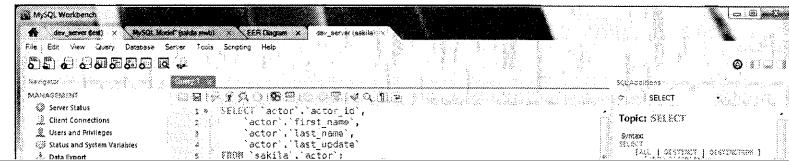
- **CHAR(M)** – символен низ с фиксирана дължина между 1 и 255 символа;
- **VARCHAR(M)** - символен низ с променлива дължина между 1 и 255 символа. Задължително се дефинира дължината при използване на типа;
- **BLOB** или **TEXT** – поле с максимална дължина 65535 символа. BLOB е за двоичен формат на данните ("Binary Large Objects"), използва се за информация с голям обем – снимки или друг тип файлове. TEXT е аналогичен. Разликата е в алгоритъма за сравнение и сортиране на съхранените данни. При BLOB той е чувствителен към малки и големи знаци (case sensitive), а при TEXT - не;
- **TINYBLOB** или **TINYTEXT** - BLOB или TEXT колони с максимална дължина от 255 символа;
- **MEDIUMBLOB** или **MEDIUMTEXT** - BLOB или TEXT колони с максимална дължина от 16777215 символа;
- **LONGBLOB** or **LONGTEXT** - BLOB или TEXT колони с максимална дължина от 4294967295 символа;
- **ENUM** – За данни от тип enumeration (избройм лист), за списък от обекти, които могат да бъдат избиранни. Лист "A" или "B" или

"C", ще изглежда ENUM ('A', 'B', 'C') и само тези стойности (или NULL) могат да бъдат избирани.

4.3.3 Инсталиране на MySQL.

За инсталацията на MySQL под Windows е необходимо да се свали или MySQL Installer, или преконфигурирана версия, която е .zip файл и може да работи като самостоятелно приложение без да се регистрира като услуга в операционната система. След инсталации сървърът се тества като се стартира от неговата /bin

MySQL Workbench е инструментът, който подпомага работата със СУБД като предоставя възможност за създаване и управление на бази от данни в средата на MySQL през потребителски графичен интерфейс.



- Импортиране на необходимите пакети;
- Регистриране на драйвер за базата от данни;
- Указване URL на базата;
- Създаване на обект от тип *Connection*.

4.4.1.1 Импортиране на пакети.

Два са основните пакета, които трябва да се заредят, като вторият от тях не е задължителен, а се използва за специфични нужди:

```
import java.sql.* ; // за стандартни JDBC приложения
import java.math.* ; // за поддръжка на BigDecimal и BigInteger
```

```
catch(ClassNotFoundException ex) {
    System.out.println("Error: unable to load driver class!");
    System.exit(1);
}
```

2. чрез *Class.forName("...").newInstance()*.

При този подход е необходимо прихващането на още два типа грешки:

```
try {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
}
catch(ClassNotFoundException ex) {
    System.out.println("Error: unable to load driver class!");
    System.exit(1);
}
catch(IllegalAccessException ex) {
    System.out.println("Error: access problem while loading!").
```

страна, при не толкова динамично използване на връзките към базата, създаването и „замрязването“ на конекции е неефективно поради причина, че операционните системи имат

За комуникация с MySQL трябва да се регистрира следното URL:

```
static final String DB_URL = "jdbc:mysql://localhost/MyDataBase";
```

4.4.2 Създаване на обект от тип *Statement*.

Statement е интерфейс, чрез който се репрезентира SQL в Java. Създава се чрез извикването на метода *createStatement()* на обект от тип *Connection*:

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

Използва се за създаване на SQL заявки. Има три вида *Statement* обекта:

- *Statement* – използва се за прости SQL заявки без параметри;
- *PreparedStatement* (Extends Statement) – използва се за прекомпилирани SQL заявки, които могат да съдържат

Подходящ за използване при изпълнение на INSERT, DELETE или UPDATE SQL заявки.

4.4.3 Обработка на резултата. *ResultSet*.

ResultSet обекта може да се разглежда като курсор или указател, който сочи към пореден ред от set-а с резултати. Методите, които той притежава, могат да се разделят в три категории:

- **Навигационни** – за придвижване на курсора;
- **За извлечане на данни** – извличане на данните от текущия ред, към който сочи курсора;
- **За определяне на валидността** – определяне на валидността на реда.

В JDBC версия 4.1, която е налична от Java SE 7 и след това, може да се използва новият блок `try` с ресурс (try-with-resources statement), който затваря автоматично обекти от тип `Connection`, `Statement` и `ResultSet`. При неуспешно затваряне ще се получи `SQLException`.

4.5 Комуникация с MySQL.

Пример 1: Създаване на база от данни.

```
import java.sql.*;

public class JDBCExample {

    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/";

    static final String USER = "root";
    static final String PASS = "root";

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try{
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
        } catch{
```

```
        if(conn!=null)
            conn.close();
        }catch(SQLException se){
            se.printStackTrace();
        }
        System.out.println("Goodbye!");
    }
}
```

Пример 2: Създаване на таблица.

```
import java.sql.*;

public class JDBCExample {

    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/STUDENTS";

    static final String USER = "username";
    static final String PASS = "password";

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try{
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Connecting to a selected database...");
```

```

try{
    if(stmt!=null)
        conn.close();
}catch(SQLException se){

try{
    if(conn!=null)
        conn.close();
}catch(SQLException se){
    se.printStackTrace();
}

System.out.println("Goodbye!");
}
}

```

Пример 3: Вмъкване на данни в таблица:

```

import java.sql.*;

public class JDBCExample {

static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
static final String DB_URL = "jdbc:mysql://localhost/STUDENTS";

static final String USER = "username";
static final String PASS = "password";

public static void main(String[] args) {
Connection conn = null;
Statement stmt = null;
try{
    Class.forName("com.mysql.jdbc.Driver");
}
}
}

```

```

stmt.executeUpdate(sql);
sql = "INSERT INTO MAGISTRI " +
      "VALUES(103, 'Angel', 'Marinov', 28)";
stmt.executeUpdate(sql);
System.out.println("Inserted records into the table...");

}catch(SQLException se){
    se.printStackTrace();
}catch(Exception e){
    e.printStackTrace();
}finally{
try{
    if(stmt!=null)
        conn.close();
}catch(SQLException se){
}
try{
    if(conn!=null)
        conn.close();
}catch(SQLException se){
    se.printStackTrace();
}
}
System.out.println("Goodbye!");
}
}

```

Пример 4: Извличане на данни от таблица.

```

import java.sql.*;

public class JDBCExample {
}

```

```

ResultSet rs = stmt.executeQuery(sql);

while(rs.next()){
    int id = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
}

rs.close();
}catch(SQLException se){
    se.printStackTrace();
}catch(Exception e){
    e.printStackTrace();
}finally{

    try{
        if(stmt!=null)
            conn.close();
    }catch(SQLException se){
    }
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){
        se.printStackTrace();
    }
}
System.out.println("Goodbye!");
}
}

```

Пример 5: Актуализиране на данни в таблица.

```

import java.sql.*;

public class JDBCExample {
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/STUDENTS";
    static final String USER = "username";
    static final String PASS = "password";

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try{

```

```

Class.forName("com.mysql.jdbc.Driver");
System.out.println("Connecting to a selected database...");
conn = DriverManager.getConnection(DB_URL, USER, PASS);
System.out.println("Connected database successfully...");
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql = "UPDATE Registration "
            + "SET age = 30 WHERE id in (100, 101)";
stmt.executeUpdate(sql);
sql = "SELECT id, first, last, age FROM Registration";
ResultSet rs = stmt.executeQuery(sql);
while(rs.next()){
    int id = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
}
rs.close();
}catch(SQLException se){
    se.printStackTrace();
}catch(Exception e){
    e.printStackTrace();
}finally{
    try{
        if(stmt!=null)
            conn.close();
    }catch(SQLException se){
    }
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){
        se.printStackTrace();
    }
}
System.out.println("Goodbye!");
}
}

```

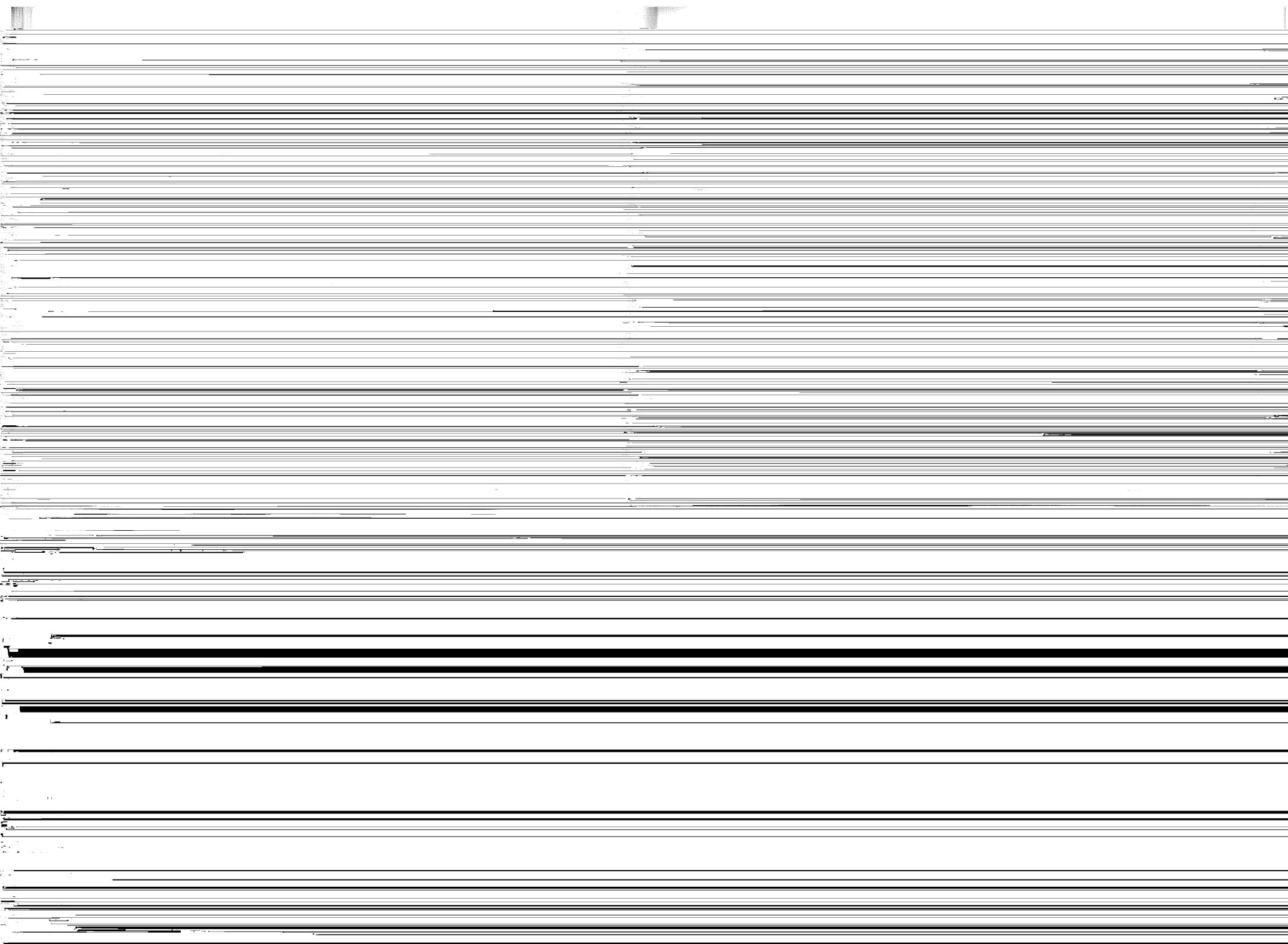
Пример 6: Изтриване на данни от таблица.

```
import java.sql.*;  
  
public class JDBCExample {  
  
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";  
    static final String DB_URL = "jdbc:mysql://localhost/STUDENTS";  
    static final String USER = "username";  
    static final String PASS = "password";  
  
    public static void main(String[] args) {  
        Connection conn = null;  
        Statement stmt = null;  
        try{  
            Class.forName("com.mysql.jdbc.Driver");  
            System.out.println("Connecting to a selected database...");
```

```
try{  
    if(conn!=null)  
        conn.close();  
}catch(SQLException se){  
    se.printStackTrace();  
}  
System.out.println("Goodbye!");  
}
```

Пример 7: Изтриване на таблица.

```
import java.sql.*;  
  
public class JDBCExample {
```



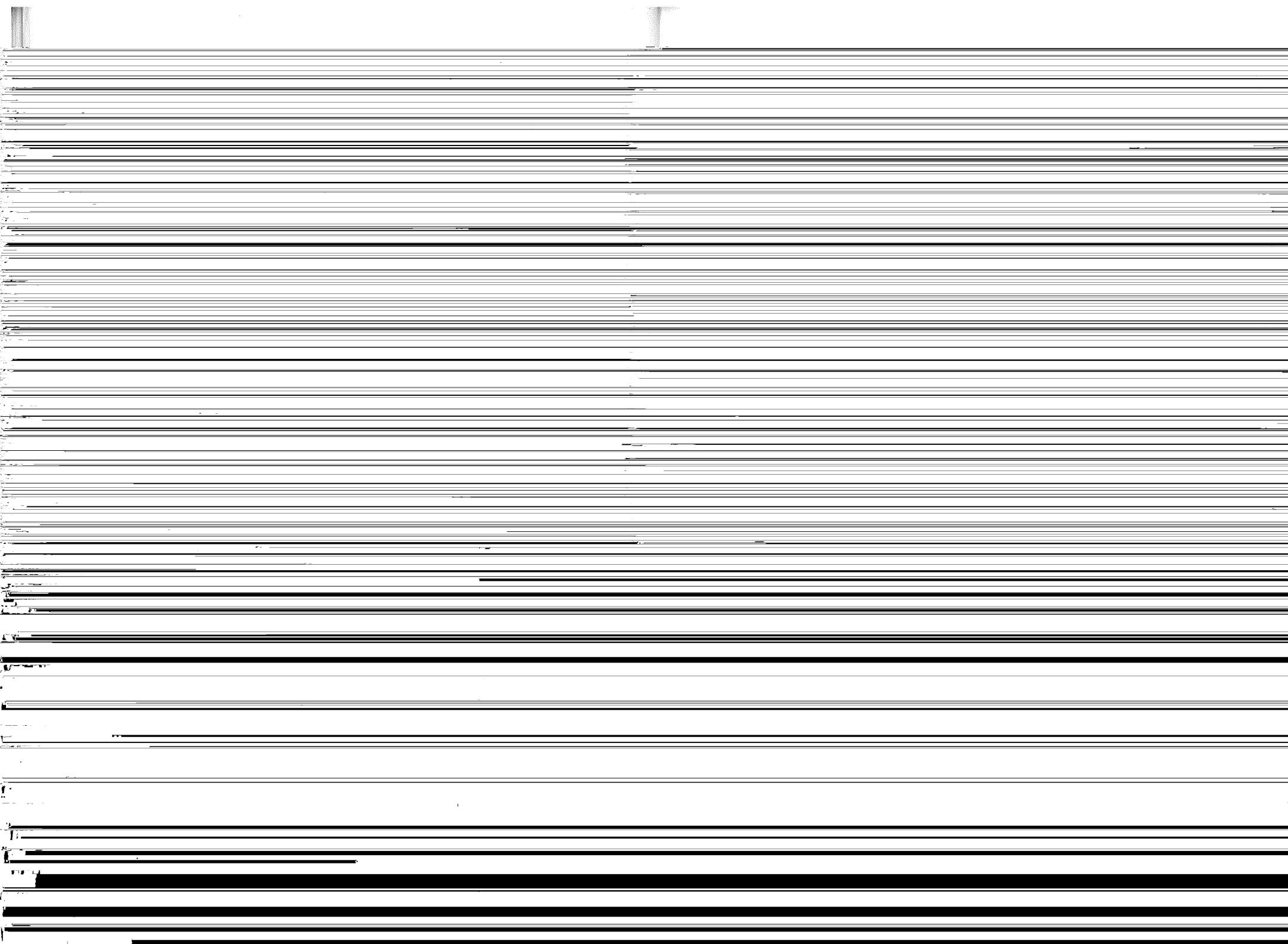
5.Web сървъри

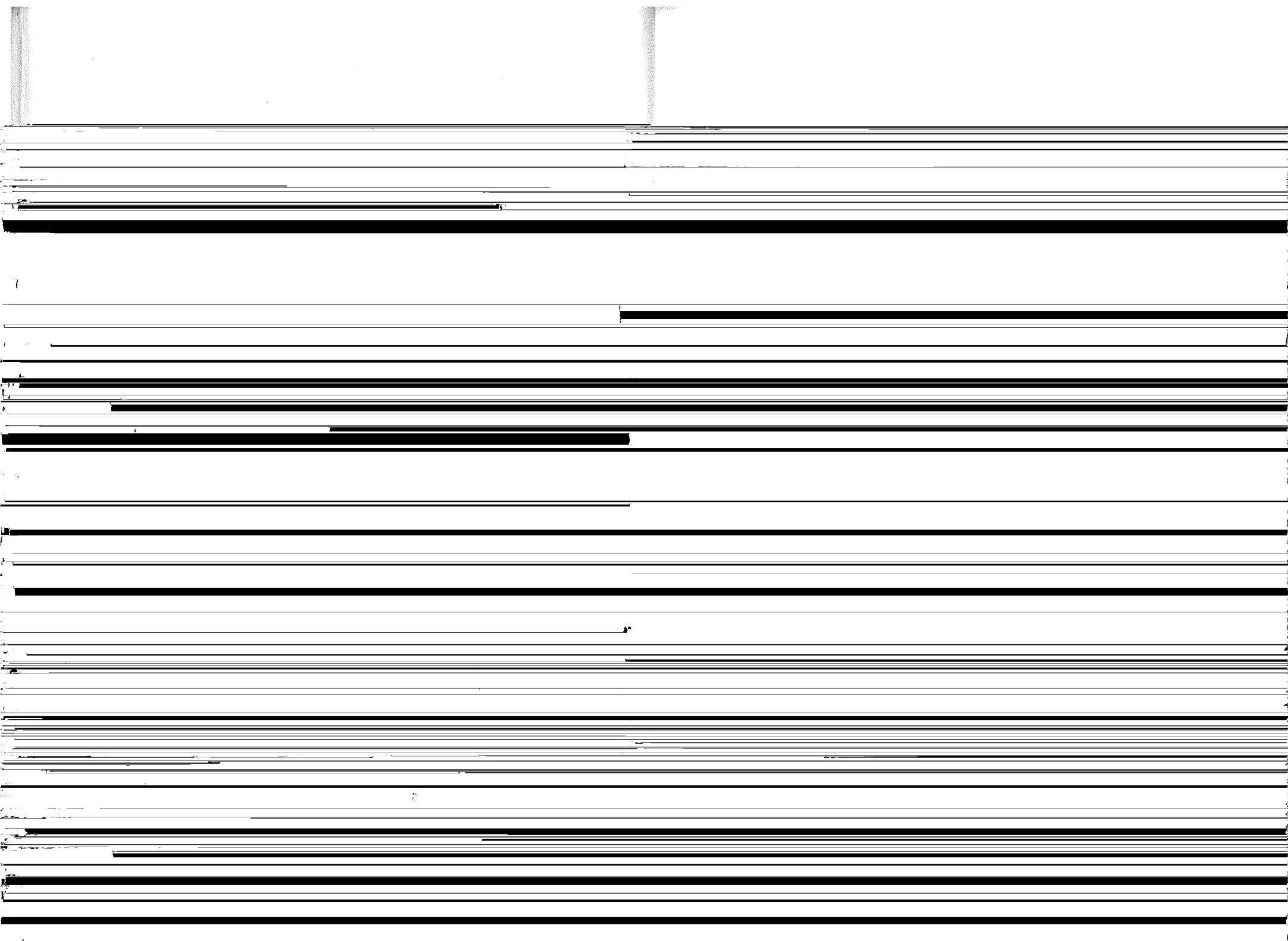
5.1 Същност на web сървъра.

Сървър (на английски: *server*) е термин, който има две тясно свързани значения:

- Компютърна програма, която предоставя услуги на други

- наличие на софтуер (уеб приложение), различен от самия сървър, който служи за генериране на динамични уеб страници;
- възможност за обслужване на множество уеб приложения;
- поддръжка на файлове с голям размер;
- контролиране на комуникацията с клиентите за предотвратяването претоварването на мрежата с цел увеличаване броя







6. Java Сървлет технология

6.1 Java сървлети.

JavaServlets са програми, които се изпълняват на уеб сървър и действат като междинен слой между заявките от уеб браузър (или друг HTTP клиент), и бази от данни или приложения, разположени на или зад сървъра. Използването на Servlets предоставя различни възможности като събиране на информация от потребителите чрез уеб форми на страници, представяне на записи от база данни или друг източник и динамично създаване на уеб страници. JavaServlets често служат за същата цел като програми, изпълнявани с помощта на Common Gateway Interface (CGI). Сървлетите предлагат няколко предимства в сравнение с CGI:

- изпълнението е значително по-добро;
- изпълняват се в рамките на адресното пространство на уеб сървъра. Не е необходимо да се създава отделен процес, за да се обслужват клиентските заявки;
- платформено независими, защото са написани на Java;
- сървлетът има достъп до пълната функционалност на библиотеките на Java. Той може да комуникира с аплети.

RMI или CORBA повикване, използване на уеб услуга, или директно изчисляване на отговор;

- изпращане на данни на клиентите (браузъри). Данните могат да бъдат в най-различни формати, включително текст (HTML или XML), двоични (GIF изображения), Excel и т.н.;
- изпращане на допълнителна мета информация на клиенти (браузъри). Това включва указване на типа документ, който ще се връща (например HTML), създаване на бисквитки, кеширане на параметри и други.

6.2 Жизнен цикъл на сървлета.

Жизненият цикъл на сървлета може да се дефинира като времето от неговото създаване до премахването му от паметта. Етапите, през които преминава сървлета, са следните:

- инициализация на сървлета чрез повикване на *init()* метода му;
- сървлетът извиква метода си *service()*, за да обслужва клиентските заявки;
- сървлетът е деактивиран чрез повикване на метода му *destroy()* от контейнера;

```
public void init() throws ServletException{
    // инициализиращ код...
}
```

Методът *service()*

Това е основният метод на сървлета, който извършва същинската работа. Сървлет-контейнерът извиква този метод, за да бъдат обслужени клиентските заявки и да бъде генериран форматиран отговор за клиента. Всеки път, когато сървлът получи заявка за сървлет, той създава нова нишка и извиква *service()*. Методът проверява типа на заявката (GET, POST, PUT, DELETE, ...) и извиква съответния метод, реализиран в сървлете.

```
public void service(ServletRequest request, ServletResponse
response) throws ServletException, IOException{...}
```

Методите *doGet()* и *doPost()* са най-често използвани методи във всяка клиентска заявка.

Методът *doGet()*:

```
public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException{
// код на сървлета
}
```

Методът *doPost()*:

```
public void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException{
// код на сървлета
}
```

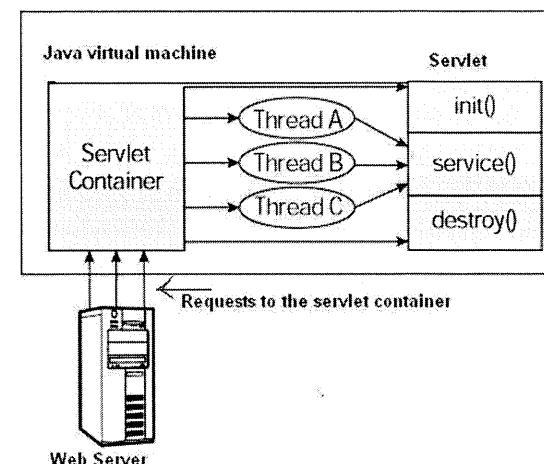
Методът *destroy()*

Методът се извиква само веднъж по време на жизнения цикъл на сървлета. Този метод дава възможност на сървлета да извърши някои финализиращи операции като затваряне на връзка към база от данни, записване на бисквитка или някаква почистваща операция. След неговото повикване сървлета е маркиран за почистване.

```
public void destroy(){
    // финализиращ код...
}
```

На фиг. 6.1 е представен типичния сценарий за жизнен цикъл на сървлета

- Получаване на HTTP заявка и делегирането и към сървлет-контейнера;
- Сървлет-контейнерът зарежда сървлета преди извикването на метода *my service()*;
- Сървлет-контейнерът прихваща множество клиентски заявки като създава множество нишки в контекста на един и същи сървлет, които да обработят заявките.



Фиг.6.1 Жизнен цикъл на сървлет.

6.3. Създаване на сървлети.

Сървлетите са джава програми, които обслужват HTTP заявки и имплементират *javax.servlet.Servlet* интерфейса. Разработчиците на уеб приложения създават сървлети като наследяват *javax.servlet.http.HttpServlet* класа - абстрактен клас, който имплементира сървлет-интерфейса и е специално проектиран за обслужване на HTTP заявки.

Примерен сървлет (*Hello Java*):

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/HelloJava")
public class HelloJava extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
    response.setContentType("text/html");
    PrintWriter out= response.getWriter();
    out.println("<h1> Hello Java </h1>");
}
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
}
}
```



Фиг.6.2 Резултат от изпълнението на „HelloJava“ сървлет.

6.4. Инсталиране на сървлет.

Според стандарта за структура и разположение, сървлетите се поставят в директорийната структура на *\WEB-INF\classes*. За да бъде достъпен за клиентски заявки, сървлетът трябва да бъде деклариран в контекста на приложението. Това става по два начина - чрез описание в дескриптора на приложението (*web.xml*) или чрез анотацията *@WebServlet*.

Декларация посредством *web.xml*:

```
<servlet>
    <servlet-name>HelloJava</servlet-name>
    <servlet-class>HelloJava</servlet-class>
```

```
</servlet>
<servlet-mapping>
    <servlet-name>HelloJava</servlet-name>
    <url-pattern>/HelloJava</url-pattern>
</servlet-mapping>
```

Идеята на дескриптора е създаване на връзка между класа на сървлета и URL-то, на което той ще отговаря. Това става на две стъпки. Първата е задаване на псевдоним на сървлета за конкретен клас и втората е обвързване на този псевдоним с определен URL.

Декларация посредством анотацията *@WebServlet*:

```
@WebServlet("/HelloJava")
public class HelloJava extends HttpServlet {
...
}
```

Тук сървлета „HelloJava“ е свързан с адреса /HelloJava. Имената на сървлетите и адресите могат да бъдат и най-често са напълно различни.

Декларация на сървлет с повече от едно URL:

```
@WebServlet(urlPatterns = {"/sendFile", "/uploadFile"})
public class UploadServlet extends HttpServlet {
...
}
```

Декларация на сървлет с допълнителна информация:

```
@WebServlet(
    name = "MyServlet",
    description = "This is my first annotated servlet",
    urlPatterns = "/processServlet"
)
public class MyServlet extends HttpServlet {
...
}
```

Декларация на сървлет с инициализиращи параметри:

```
@WebServlet(
    urlPatterns = "/imageUpload",
    initParams =
    {
        @WebInitParam(name = "saveDir", value = "D:/FileUpload"),
        @WebInitParam(name = "allowedTypes", value =
        "jpg,jpeg,gif,png")
    }
)
```

```
    }
}

public class ImageUploadServlet extends HttpServlet { ... }
```

6.5. Обработване на клиентски заявки. Методите *doGet()* и *doPost()*.

Обработване на GET заявка, HTML форма:

```
<html>
<body>

    <form action="HelloForm" method="GET">
        First Name: <input type="text" name="first_name"> <br />
        Last Name: <input type="text" name="last_name" /> <input
                    type="submit" value="Submit" />
    </form>
</body>
</html>
```

Сървълт, обработващ заявката от формата чрез метода *doGet()*:

```
@WebServlet("/HelloForm")
public class HelloForm extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
```

```
First Name: <input type="text" name="first_name"> <br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

Сървълт, обработващ заявката от формата чрез метода *doPost()*:

```
@WebServlet("/HelloForm")
public class HelloForm extends HttpServlet {
    public void doPost(HttpServletRequest request,
                      HttpServletResponse response) throws ServletException,
                      IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Using POST Method to Read Form Data";
        String docType = "<!doctype html public "-//w3c//dtd
                        html 4.0 "
        + "transitional//en//">\n";
        out.println(docType + "<html>\n" + "<head><title>" +
                    title + "</title></head>\n" +
                    "<body bgcolor=\"#f0f0f0\">\n" +
                    "<h1 align=\"center\">" + title + "</h1>\n" +
                    "<ul>\n" +
                    "    <li>First Name: " + request.getParameter("first_name") + "</li>\n" +
                    "    <li>Last Name: " + request.getParameter("last_name") + "</li>\n" +
                    "</ul>\n" +
                    "</body>\n" +
                    "</html>");
    }
}
```

7. Създайте необходимите сървлети за организиране на CRUD-операции върху базата от данни „Студент“ от предходното упражнение.

7. Java Server Pages (JSP)

7.1 Същност.

Java Server Pages (JSP) е технология за разработка на уеб страници, които поддържат динамично съдържание. Технологията представлява възможност за разработчиците да вмъкнат Java код в HTML страници чрез използване на специални тагове.

JavaServer Pages е вид Java сървлет, който е предназначен да изпълнява ролята на потребителски интерфейс за уеб приложения. Уеб разработчиците пишат JSP като текстови файлове, в които се комбинират HTML или XHTML код, XML-елементи и вградени действия и команди на Java.

Използването на JSP позволява събиране на информация от потребителите чрез уеб форми, представяне на записи от база данни или друг източник, създаване на динамични уеб страници.

JSP тагове могат да бъдат използвани за различни цели, като например извличане на информация от база данни или регистриране на предпочтенията на потребителите, достъп до JavaBeans компоненти, предаване на контрол между страници и споделяне на информация между заявки, страници и т.н.

Предимства на JSP пред подобни технологии

- В сравнение с Active Server Pages (ASP) - предимствата на JSP са в две посоки. Първо, динамичната част е написана на Java, не Visual Basic или друг MS конкретен език, така че е по-мощен и лесен за използване. Второ, тя е преносима към други операционни системи и уеб сървъри, които не са продукти на Microsoft;
- В сравнение със Servlets - по-удобно е да се пише (и модифицира) обикновен HTML, отколкото да се използват множество println() методи, които генерират HTML;
- В сравнение със Server-Side Includes (SSI) - SSI е предназначен само за прости включения, а не за "истински" приложения, които използват данни от формуляри, осъществяват връзки с бази от данни и други;
- В сравнение с JavaScript - JavaScript може да генерира HTML динамично на клиента, но не може да си взаимодейства с уеб

сървър, за да изпълнява сложни задачи като достъп до бази от данни и обработка на изображения и т.н.

- В сравнение HTML – HTML не може да генерира динамично съдържание.

7.2 JSP процес.

Използването на JSP като обработващ ресурс преминава през следните етапи:

- Както и при нормална страница, браузърът изпраща заявка за HTTP към уеб сървър;
- Уеб сървърът идентифицира, че желания ресурс е JSP страница и препраща заявката към JSP енджина.

Това прави процеса по-ефективен отколкото при други скриптови езици (като PHP) и следователно по-бърз.

В известен смисъл, JSP страницата е просто още един начин да се напише сървлет, без да се налага на човек да бъде способен да програмира на Java. С изключение на етапа на транслация от JSP страница към сървлет, през останалата част от съществуването си тя се обработва точно като обикновен сървлет.

7.3. Жизнен цикъл на JSP.

JSP жизненият цикъл може да се дефинира като целия процес от

Четирите основни фази на JSP жизнения цикъл са много сходни с този на сървлета и те са както следва:

7.4. JSP синтаксис, тагове и елементи.

Скриплети

Скриплетът може да съдържа произволен брой JAVA декларации, променливи, методи или изрази, които са валидни в скриптовите езици:

```
<% code fragment %>
```

Всеки текст, HTML таг или JSP элемент трябва да са извън scriptlet таговете:

```
<html>
  <head>
    <title>Hello Java</title>
  </head>
  <body>
    Hello Java! <br />
    <% out.println("Your IP address is " +
request.getRemoteAddr());%>
  </body>
</html>
```

JSP декларации

Посредством тях се декларираят една или повече променливи или методи, които щат достъпни за използване по-късно във файла

JSP изрази

Елементът за JSP израз съдържа код, характерен за скриптовите езици, който се обработва и изходът му се конвертира в стринг. Тъй като стойността на израза се превръща в низ, може да се използва израз в рамките на един ред от текст, независимо дали е маркиран с HTML таг, в JSP файл. Елементът за израз може да съдържа всеки текст, който е валиден според езиковата спецификация на Java, но не може да се използва точка и запетая, за да се сложи край на израза. Синтаксисът е следният:

```
<%= expression %>
```

Пример:

```
<html>
  <head>
    <title>JSP Expression</title>
  </head>
  <body>
    <p>
      Today's date:
      <%= (new java.util.Date()).toLocaleString()%>
    </p>
  </body>
</html>
```

JSP директиви

JSP директивата засяга цялостната структура на класа на сървлета. Тя обикновено има следния вид:

JSP имплицитни обекти

JSP поддържа девет обекта, които автоматично се дефинират и се наричат имплицитни. Това са:

- *request* - HttpServletRequest обект, свързан със заявката на клиента;
- *response* - HttpServletResponse обект, свързан с отговора към клиента;
- *out* - PrintWriter обект, свързан с изпращането на отговора към клиента;
- *session* - HttpSession обект, свързан със заявката на клиента;
- *application* - ServletContext обект, свързан с контекста на приложението;
- *config* - ServletConfig обект, свързан със страницата;
- *pageContext* - капсулира използване на специфични сървърни функции, като например изпълнението на по-сложни JspWriters;
- *page* - синоним, който се използва, за да се извикват методите, дефинирани в новополучения сървъл клас;
- *Exception* - обект, който позволява данните за изключения да бъдат достъпни чрез определен JSP.

JSP действия

JSP действията използват конструкции в XML синтаксис, за да контролират поведението на сървъл-енджина. Позволяват събития като например динамично вмъкване на файл, повторно

- *jsp:include* – включва файл в момента, в който страницата бъде заявлена;
- *jsp:useBean* – намира съществуващ или подава заявка за създаване на JavaBean компонент;
- *jsp:setProperty* – задава характеристика на JavaBean компонент;
- *jsp:getProperty* – вмъква характеристика на JavaBean компонент в генерирания отговор;
- *jsp:forward* – пренасочва клиента към друга страница;
- *jsp:plugin* – генерира браузър-специфичен код, който създава OBJECT или EMBED тага за Java пъгън;
- *jsp:element* – динамично създава XML елементи;
- *jsp:attribute* – определя атрибут на динамично създаден XML елемент;
- *jsp:body* – дефинира тяло (body) на динамично създаден XML елемент;
- *jsp:text* – използва се за шаблонни записи на текст в JSP страници или документи.

JSP осигурява пълните възможности на Java да бъдат вградени в учеб приложение. Могат да се използват всички API-и конструкции на Java в JSP програмирането, включително проверки на условия, цикли и други.

Операторът "If":

```
<%! int day = 3; %>
<html>
  <head>
    <title>IF...ELSE Example</title>
  </head>
```

```
    }
%>
</body>
</html>
```

JSP дава възможност за използването на всички основни цикли от Java.

Пример за „for“ цикъл:

```
<%!int fontSize;%>
<html>
<head>
<title>FOR LOOP Example</title>
</head>
<body>
<%
    for (fontSize = 1; fontSize <= 3; fontSize++) {
%>
<font color="green" size="<%fontSize%>"> JSP Tutorial
</font>
<br />
<%
    }
%>
</body>
</html>
```

Пример за „while“ цикъл:

```
<%!int fontSize;%>
<html>
<head>
<title>WHILE LOOP Example</title>
</head>
<body>
<%
    while (fontSize <= 3) {
%>
<font color="green" size="<%fontSize%>"> JSP Tutorial
</font>
<br />
<%
    }
%>
```

Въпроси и задачи:

1. Какво е общото и различното между JSP и Servlet технологиите?
2. Създайте информационна система „Студент“, използвайки създадената вече база от данни и Java EE технологиите – JSP, Servlet.
3. Какви проблеми при изпълнението могат да възникнат?
4. Каква е разликата между *Forward* и *Redirect*?

Литература:

1. Antonio Goncalves „Beginning Java™ EE 6 Platform with GlassFish™ 3 - From Novice to Professional“ ISBN-13 (pbk): 978-1-4302-1954-5, 2009
2. Bogdan Ciubotaru, Gabriel-Miro Muntean, “Advanced Network Programming – Principles and Techniques” Springer London ISBN 978-1-4471-5291-0, 2013г.Jan Graba, “An Introduction to Network Programming with Java” Springer London ISBN 978-1-4471-5253-8, 2013
3. Elliotte Rusty Harold, “Java Network Programming, Fourth Edition” O'Reilly ISBN: 978-1-449-35767-2, 2014
4. Kameron Cole, Robert McChesney, Richard Raszka, „Advanced Java EE Development for Rational Application Developer 7.5: Developers' Guidebook“, ISBN: 978-1-931182-31-7, 2011
5. Peter A. Pilgrim, “Java EE 7 Developer Handbook”, ISBN 9781849687942, 2013