



Технически университет – Варна
Катедра “Компютърни науки и технологии”

Христо Божидаров Ненов



МРЕЖОВО ПРОГРАМИРАНЕ С **JAVA**

Ръководство за лабораторни упражнения
Варна, 2016

**Технически университет – Варна
Катедра “Компютърни науки и технологии”**

Христо Божидаров Ненов

МРЕЖОВО ПРОГРАМИРАНЕ С JAVA

Ръководство за лабораторни упражнения

**Варна
2016**

МРЕЖОВО ПРОГРАМИРАНЕ С JAVA

Автор: © Христо Божидаров Ненов

I S B N 978-954-20-0756-2

Съдържание:

1.	Входно-изходни потоци в Java.....	6
1.1	Потоци за вход и изход в Java.....	6
1.2	Типове потоци.....	6
1.3	Влагане на класове.....	12
1.4	Автоматично затваряне на потоци. Използване на „try“ с ресурс. Интерфейсът <i>AutoCloseable</i>	14
1.5	Разбиване на символни потоци на отделни елементи.....	15
1.6	Сериализация.....	18
2.	Многозадачност в Java.....	22
2.1	Същност.....	22
2.2	Нишки в Java.....	22
2.2.1	Наследяване на класа Thread.....	23
2.2.2	Имплементиране на интерфейса Runnable	24
2.3	Многонишков сървър.....	25
2.4	Взаимно блокиране. Locks и Deadlock.....	29
2.5	Синхронизиране на нишки.....	31
2.6	Неблокиращи сървъри.....	37
2.6.1	Реализация на NIO.....	38
3.	Мрежови потоци.Сокети. Класът iNet.....	44
3.1	Класът INet.....	44
3.2	Използване на сокети.....	45
3.2.1	TCP/IP сокети.....	46
3.2.2	Datagram (UDP) Сокети.....	51
3.3	Използване на графичен интерфейс.....	56
4.	JDBC. Работа с бази от данни.....	60
4.1	Същност.....	60
4.2	Архитектура на JDBC.....	60
4.3	Система за управление на бази от данни.....	62
4.3.1	MySQL.....	63
4.3.2	Данниови типове в MySQL.....	65
4.3.3	Инсталиране на MySQL.....	68
4.4	Комуникация с база от данни.....	69
4.4.1	Създаване на връзка към базата.....	69
4.4.1.1	Импортиране на пакети.....	70
4.4.1.2	Регистрация на драйвер.....	70
4.4.1.3	Задаване URL на базата.....	72
4.4.1.4	Създаване на обект от тип Connection.....	73

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
4.5 Комуникация с MySQL.....	76
5. Web сървъри.....	86
5.1 Същност на web сървъра.....	86
5.2 Apache Tomcat.....	87
5.3 Java уеб приложения.....	88
6. Java сървлет технология.....	94
6.1 Java сървлети.....	94
6.2 Жизнен цикъл на сървлета	95
6.3. Създаване на сървлети	97
6.4. Инсталациране на сървлет.....	98
6.5. Обработване на клиентски заявки. Методите <i>doGet()</i> и <i>doPost()</i>	100
7. Java Server Pages (JSP).....	103
7.1 Същност.....	103
7.2 JSP процес.....	104
7.3. Жизнен цикъл на JSP.....	105
7.4. JSP синтаксис, тагове и елементи.....	106
Литература:	112

Предговор

Ръководството за лабораторни упражнения по дисциплината „Мрежово програмиране с Java“ е предназначено за студентите от ОКС „Магистър“ специалност „Компютърни мрежи и комуникации“. Дисциплината използва като база знанията на студентите от предходни дисциплини като „Обектно ориентирано програмиране“, „Обектно ориентирани приложения“, „Програмни технологии в Интернет“, „Компютърни мрежи“, „Бази от данни“, „Операционни системи“. Целта на дисциплината е да предаде знания на студентите в областта на мрежовото програмиране, базирано на Java технологии. Разделите от мрежовото програмиране, които са разгледани в ръководството, са „сокет“ комуникация между приложения и възможности на Java за изграждане на сложни корпоративни информационни системи.

Като програмни инструменти, за решаване на поставените задачи, се използва JavaSE (v.1.7), както и технологиите за web програмиране – Servlet и JSP, които са част от Java EE спецификацията. Разглеждат се още дизайн и работа с бази от данни, както и технологии за комуникации на Java приложения със системи за управление на бази от данни (СУБД).

(Изборът на версия на Java v.1.7 е продиктуван от факта, че към момента на създаване на ръководството актуалната версия на Java EE е 1.7).

Упражненията са структурирани в два раздела – сокет мрежова комуникация и Java web технологии. Етапите, през които се преминава, дават възможност на студентите да разгледат аспектите на мрежовото програмиране, базирано на Java технологии и в края на курса на обучение да достигнат до завършена напълно функционираща система.

Авторът изказва благодарности на рецензентите доц. д-р инж. Н. Рускова и доц. д-р инж. Н. Николов за забележките и предложенията към ръководството, както и на ас. С. Попова, и на Георги „Жорето“ Цветков за оказаното съдействие при изготвянето му.

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 „произвежда“ информация, или „консумира“ такава. Системата на Java виртуалната машина (JVM) за I/O свързва тези абстракции с конкретни физически устройства. Всички потоци в Java имат едно и също поведение, независимо от различните физически устройства към които се свързват. По този начин едни и същи I/O класове могат да се прилагат за различни типове устройства. Това означава, че потокът за вход може да е абстракция за различни видове вход: от файл, от клавиатура или от мрежови сокет. Аналогично е и с потоците за изход. Те могат да се отнасят за конзола, за файл или за мрежова връзка. Потоците за вход и изход в Java са изключително елегантен подход за решаване на I/O процеси, тъй като една съвсем малка част от програмния код, написан от програмиста, се интересува от това към какво реално устройство ще бъде свързан. За останалата част от програмата няма разлика дали се чете от клавиатура или от мрежова връзка. Повечето от класовете за работа с потоци в Java са разположени в пакета *java.io*.

1.2 Типове потоци.

Java дефинира различни групи от типове потоци в зависимост от техните специфики и възможности:

- Байтово-ориентирани потоци (Byte Streams) - I/O на двоична информация;

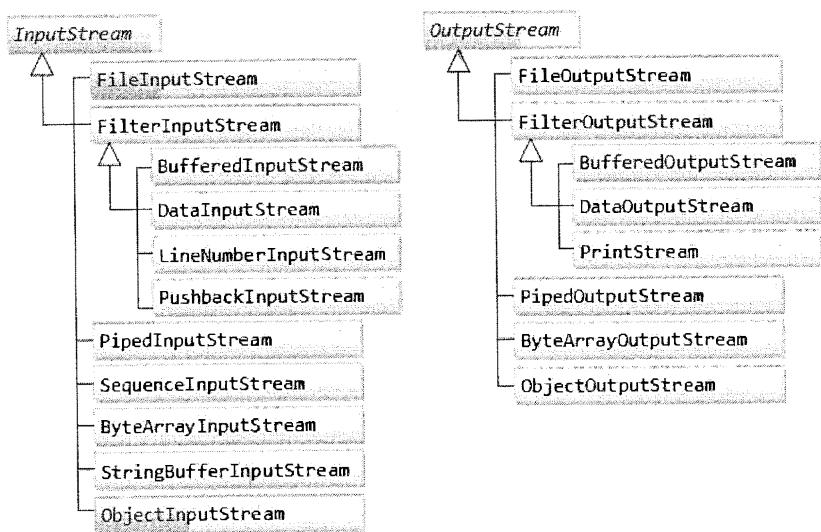
- Символно-ориентирани потоци – интерпретират информацията като символи;
- Буферирани потоци (Buffered Streams) – оптимизиране на работата чрез редуциране на обръщенията към системните API;
- Потоци за форматирани данни (Scanning and Formatting) – позволяват четене и запис на форматиран текст;
- Стандартни 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;
- Обектно-ориентирани потоци (Object Streams) – извършват байтово четене и запис на обекти.

По отношение на това как се интерпретират данните, с които те работят, потоците в Java се делят на два основни типа: *байтово-ориентирани* и *символно-ориентирани*.

Байтово-ориентирани потоци (двоични).

Байтовият поток осигурява удобно средство за организиране на входно/изходни операции с данни, които се интерпретират като байтове. Този тип поток се използва при четене/запис на двоична информация.

Байтово-ориентиранныте потоци са дефинирани в Java чрез две йерархии от класове. На върха на тези йерархии са двата абстрактни класа: *InputStream* за вход и съответно *OutputStream* за изход. Всеки от тези класове има наследници, които отговарят за връзката с конкретни физически устройства, файлови системи, мрежови връзки, буфери в паметта и др.:



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 онаглеждане 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 проверка дали всеки един от потоците съдържа референция към обект преди да бъде извикан методът `close()`.

Твърдото залагане на имена на файлове не е добра практика в Java програмирането. Това прави кода зависим от операционната система, тъй като начина на задаване на път до файл е различен за различните операционни системи. Съществуват различни техники, с които се заобикаля този проблем, но за по-добро онаглеждане на основните идеи в примерите те няма да бъдат разглеждани.

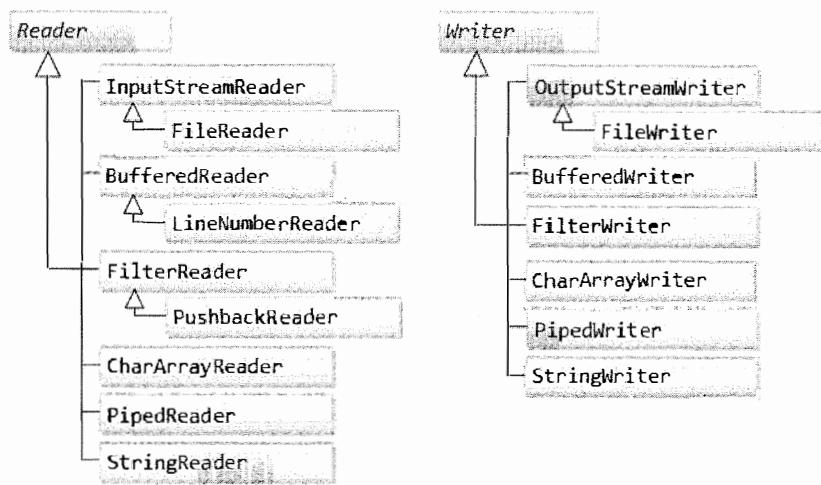
Байтово-ориентираните потоци представляват I/O на ниско ниво. Използването на такъв вид поток се препоръчва само при копиране на двоична информация. За повечето от другите случаи са по-подходящи символно-ориентираните потоци или други потоци от още по-високо ниво, каквито Java притежава.

Символно ориентирани потоци.

Поради факта, че байт-ориентираните потоци са неудобни за обработка на информация, съхранявана в Unicode (Unicode използва два байта), има отделна йерархия на класове за обработка на Unicode символи. Двата абстрактни класа, които стоят най-отгоре на йерархията на символно-ориентирани потоци, са `Reader` и `Writer`. Класовете от тази йерархия извършват операции четене и запис,

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-ориентирани потоци са по-ефикасни от байтово-ориентирани. 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-ориентирани потоци да бъдат забранени.



Фиг.1.2 Йерархия на класовете за работа със символно-ориентирани потоци.

На ниско ниво, символните потоци са също потоци от байтове, но те представляват ефикасен и удобен инструмент за интерпретация на информация под формата на символи

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");
            outputStream = new FileWriter("output.txt");

            int c;
            while ((c = fin.read()) != -1) {
                fout.write(c);
            }
        } finally {
            if (fin != null) {
                fin.close();
            }
            if (fout != null) {
                fout.close();
            }
        }
    }
}

```

Примерът "CopyCharacters" е аналогичен на примера „CopyBytes“ с тази разлика, че тук са използвани класове за работа със символно-ориентирани потоци. Причината, поради която обектът, способстващ четенето от потока да е деклариран като тип `int` е, че абстрактният метод в класа "Reader", който се наследява, е от целочислен тип: `int read()`.

1.3 Влагане на класове.

Многообразието от класове за работа с потоци в Java, дава възможност на програмистите да извършват процеси отвъд обикновеното писане и четене на информация. Възможностите,

които съществуват, са по отношение на интерпретиране и трансформиране на данните, които проличат 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.

```

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class FileInput {

    public static void main(String[] args) {

        File file = new File("input.txt");
        FileInputStream fis = null;
        BufferedReader bis = null;
        DataInputStream dis = null;

        try {
            fis = new FileInputStream(file);
            // използване на BufferedReader за бързо четене
            bis = new BufferedReader(fis);
            dis = new DataInputStream(bis);
            // dis.available() връща 0 ако файлът няма повече редове
            while (dis.available() != 0) {
                System.out.println(dis.readLine());
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            // затваряне на потоците и освобождаване на ресурси
            fis.close();
            bis.close();
            dis.close();
        }
    }
}

```

```
        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 автоматично затварян, то той трябва да имплементира интерфейсът `"AutoCloseable"`. Всички потоци в Java имплементират този интерфейс и имат възможност да използват неговото предимство. Това се осъществява като съответния поток или потоци биват декларириани като ресурси към блока `"try"`. Това сваля ангажимента от програмиста за затваряне на потоците. Единствената задача, която остава, е обработката на изключение/я, които все пак е възможно да се получат.

Пример 4. Използване на „try“ с ресурс.

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class ReadFromFileNew {
    public static void main(String[] args) {

        try (BufferedReader bf = new BufferedReader(
                new FileReader("proba.txt")) ) {

            String str;
            while ((str = bf.readLine()) != null)
```

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

1.5 Разбиване на символни погоци 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 изисква 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:

- *StringTokenizer* – клас за работа със символни низове, при който има възможност даден символен низ да бъде разделен на елементи (подстрингове) спрямо разделител, зададен от програмиста. Отделните елементи, които се получават в следствие на работата на класа, могат да бъдат интерпретирани чрез методи като различни данни типове;
 - *String.split()* – метод на класа *String* за разделяне на подстрингове, работещ на базата на регулярен израз. Връща обект от тип масив от стрингове *String[]*.
 - *Scanner* – пакет за работа с потоци, обхващащ процеса от достъпа до ресурса до интерпретирането на данните.

В зависимост от спецификата на решаваната задача, всеки от изброените методи има своите предимства и недостатъци.

Към настоящия момент Java програмистите изразяват своето предпочтение към използването на Scалпъг, тъй като той съдържа в себе си голям набор от функционалности. Това от своя страна изисква добро познаване на програмния му интерфейс.

От друга страна, въпреки че се счита за вече устаряла 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, неговият клас трябва да имплементира интерфейсът `Serializable`. Това е „маркер“ интерфейс и няма полета или методи, които да се имплементират. Единствената му роля е да укаже на компилатора, че въпросният клас трябва да има възможност за сериализация. Процесът по сериализация се осъществява от класовете `ObjectInputStream` и `ObjectOutputStream`. Ако даден обект не трябва да бъде сериализиран, той се декларира с ключовата дума – `transient`.

Пример 4: Сериилизация и десериализация на обект.

Person.java

```

import java.io.Serializable;
public class Person implements Serializable {

```

```

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;
}

public String getFirstName() {
    return firstName;
}

@Override
public String toString() {
    return "Person{" + "firstName='" + firstName + '\'' +
    ", middleName='" + middleName + '\'' +
    ", lastName='" + lastName + '\'' + '}';
}
}

```

SerializationDemo.java

```
import java.io.*;  
  
public class SerializationDemo {  
    public static void main(String[] args) {  
        String filename = "file.ser";  
  
        Person p = new Person("Chan", "Kong", "Sang");  
  
        FileOutputStream fos = null;  
        ObjectOutputStream out = null;  
        try {  
            fos = new FileOutputStream(filename);  
            out = new ObjectOutputStream(fos);  
            out.writeObject(p);  
  
            out.close();  
        } catch (Exception ex) {  
            ex.printStackTrace();  
        }  
  
        FileInputStream fis = null;  
        ObjectInputStream in = null;  
        try {  
            fis = new FileInputStream(filename);  
            in = new ObjectInputStream(fis);  
            p = (Person) in.readObject();  
            in.close();  
        } catch (Exception ex) {  
            ex.printStackTrace();  
        }  
        System.out.println(p);  
    }  
}
```

Въпроси и задачи:

1. Какво представляват потоците в Java и какви са практиките за работа с тях?
2. Какви видове потоци има?
3. Защо е необходимо разбиване на поднизове при работа със символни низове?
4. Изпълнете примерите от упражнението и анализирайте резултатите.

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

Код на служител: 1 – мениджър, 2 – работник. Трудов стаж в брой години. Заплащане: променлива от тип *int* с нулема стойност – 0. Създайте програма, която прочита файла и създава обекти от тип „мениджър“ или „работник“ в зависимост от кода. Начислява заплащане по произволна формула, сериализира създадените обекти във файл и архивира полученият файл в *.zip* формат. Полученият архив се отваря, десериализира и данните се извеждат на екрана. При избор на архитектура на класовете, да се отчетат принципите на обектно-ориентираното програмиране.

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 програмиране без използването на технология за едновременното извършване на множество задачи и процеси. Една от технологията, които предоставят такава възможност, е технологията на нишките.

Какво представлява нишката? От гледна точка на операционните системи, нишката представлява под-процес, част от един по-голям процес. С други думи, нишката е част от контекста на даден процес (чат от неговото адресно пространство в паметта). Следствие на този факт е основната разлика между процес и нишка, а именно нишките използват общо работно пространство и могат да споделят информация помежду си, докато процесите имат индивидуално (различно) работно пространство и за пренос на информация от един процес към друг трябва специална технология, предвидена за това. От гледна точка на програмния код, нишката е клас със специално предназначение и дефинирано действие. Това действие може да бъде стартирано, изпълнено и съответно спряно в рамките на изпълнението на главния клас (главния процес).

2.2 Нишки в Java.

Дори и при наличието на мултипроцесорен хардуер, операционните системи имат нужда от стратегия за определянето на коя точно нишка от множеството да бъде разрешено заемането на процесора и време за изпълнение. Два са основните фактора за това:

- приоритет на нишките (В Java е число от 0 до 10, като 10 е максималния приоритет. По подразбиране приоритета на нишка в Java е 5);
- график на изпълнение на нишките, чрез стандартно времеделение (**pre-emptive**), или базирано на логиката на кода (**cooperative**).

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 реализацията на многозадачност, без да се налага обръщение към някакъв програмен интерфейс или системно извикване на операционната система.

Създаване на нишка в Java се осъществява по два начина:

- чрез наследяване на класа *Thread*;
- чрез имплементиране на интерфейса *Runnable*.

И в двата случая резултатът е клас, чийто основен метод “*run*” реално дефинира какво ще се случи когато бъде стартирана нишката. Наличието на два подхода може да се определи най-общо със следното: ако даден клас в Java е самостоятелен (не е наследник) и той трябва да бъде нишка, тогава може да се използва подходът с наследяване на класа *Thread*. Ако класът обаче е част от класова иерархия (наследява родителски клас), тогава поради забраната за множествено наследяване в Java следва да се избере случая с имплементиране на интерфейса *Runnable*.

Има разбира се и други съществени разлики, които са част от „конкурентното програмиране“ в Java и не са обект на изучаване в текущата дисциплина.

2.2.1 Наследяване на класа *Thread*.

Както вече знаем, методът “*run*” е този, който определя действието на нишката. Той представлява нейния главен (*main*) метод. Също както и главните методи на обикновените класове той не може да бъде извикан директно. Програмата, която иска да се обърне към него, извиква методът *start()* от класа *Thread*, който пък се обръща към методът “*run*”.

Класът *Thread* има седем конструктора, но се използват предимно два от тях:

- *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>)*
- *Thread(Runnable<object>, String<name>)*

Например:

```
public class RunnableShowName implements Runnable {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new RunnableShowName());
        Thread thread2 = new Thread(new RunnableShowName());
```

Друг начин е да се декларират нишките като полета на класа.

Например:

```
public class RunnableHelloCount implements Runnable {
    private Thread thread1, thread2;
    public static void main(String[] args) {
```

```
    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 сървърни програми, познати досега - те могат да се справят само с една връзка в даден момент. Това ограничение е недопустимо за повечето реални приложения и би направило софтуера безполезен. Има две възможни решения:

- използването на сървър с не-блокиращ принцип на действие;
- използването на многонишков сървър.

Преди Java 1.4 не е имало спецификация, предвидена да работи с неблокиращи входно-изходни (I/O) операции. Единственият възможен подход за Java програмистите е бил многонишковия сървър. Въвеждането на неблокиращите I/O в 1.4 е огромна стъпка в развитието на Java като език за мрежово програмиране.

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 );
            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 message"

```

```

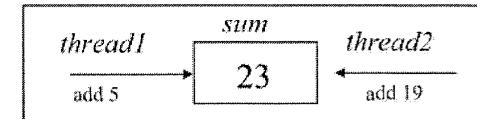
('QUIT' to exit): " );
            message = userEntity.nextLine();
            output.println( message );
            response = input.nextLine();
            System.out.println( "SERVER> " +
                response );
        } while ( !message.equals( "QUIT" ) );

        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*“.

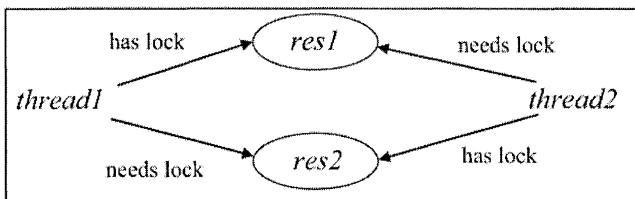


Фиг.2.1 Достъп до общ ресурс.

За да актуализират „*sum*“, всяка нишка ще трябва да изпълни следната серия от по-малки операции: да прочете текущата стойност на „*sum*“, да създаде копие от него, да добави

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 единия случай резултатът ще бъде 28 (23+5), в другия 42 (23+19). Проблемът е, че под-операции от двете актуализации могат да се припокрият една с друга. За да се избегне този проблем в Java, може да се изиска нишката да блокира обекта, който трябва да се актуализира. Само нишката, която е извършила блокирането, може да актуализира обекта. Всяка друга нишка трябва да изчака докато обекта бъде отблокиран. След като първата нишка завърши работата си с блокирианият ресурс, следва да го освободи, което го прави достъпен за другите нишки (Трябва да се има предвид, че нишки, които изискват само четене, не се нуждаят от изчакване за освобождаване на обекта!).

Една лоша възможност в подобна ситуация е получаване на взаимно блокиране, така наречения ***deadlock***. Състояние на взаимно блокиране се случва когато нишки чакат за събития, които никога няма да се случат. Да разгледаме пример, показан на фиг. 2.2.



Фиг.2.2 Взаимно блокиране.

Тук *thread1* има заключване на ресурс *res1*, но е необходимо да се получи достъп до ресурс *res2*, за да завърши обработката ѝ (Така че да може да освободи блокировката на *res1*). В същото време обаче, *thread2* заключва *res2*, но е необходимо да получи достъп до *res1*, за да завърши неговата обработка. За съжаление, само добър дизайн може да предотврати подобни ситуации.

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:

```
public synchronized void updateSum(int amount) {
    sum+=amount;
}
```

Ако ресурсът не е заключен когато се извика описаният по-горе метод, тогава той се заключва като се предотвратява опита на всяка друга от изпълняващите се нишки да осъществи достъп до метода. Всички други нишки, които искат да извикат този метод трябва да изчакат. След като методът е приключил изпълнението си, обектът се освобождава и се предоставя на другите нишки. Ако един обект има повече от един синхронизиран метод, свързан с него, във всеки даден момент само един от тях може да бъде активен. За повишаване на ефективността и избягване на взаимни блокировки се използват следните методи:

- *wait()*;
- *notify()*;
- *notifyAll()*

Ако една нишка, изпълняваща синхронизиран метод, не е в състояние да продължи своята работа, тогава тя може да премине в състояние на изчакване като извика методът *wait()*. Това освобождава заключения обект и позволява на други нишки да получат достъп до него. Извикването на *wait()* обаче може да доведе до изключението *InterruptedException*, което или трябва да бъде прихванато, или методът трябва да бъде деклариран с *throw*. Когато синхронизираният метод завърши своята работа, той може да извика методът *notify()*, с което да „събуди“ друга чакаща нишка. При условие, че чакащите нишки са повече от една, тогава се използва метода *notifyAll()*. В тази ситуация или различния приоритет на нишките ще определи реда на тяхното изпълнение, или Java виртуалната машина ще вземе решение за това.

Методите `wait()`, `notify()` и `notifyAll()` може да се извикват само от нишката, която е извършила зачленяването на обекта (т.е., в рамките на синхронизирания метод или в рамките на друг метод, който е извикан от синхронизирания). Ако някой от изброените методи се извика от друго място, ще се генерира изключение `IllegalMonitorStateException`.

Пример 2 е класическият проблем производител - консуматор, в които даден производител, генерира копия на известни ресурси (коли на производствена линия, шоколадови бонбони, дървени столове в дърводелски цех или др.) и потребител, който премахва копията на ресурса. Въпреки че това до голяма степен е теоретичен пример за услуга, която може да бъде предоставена посредством сървърна програма, той може да бъде модифициран. Например сървър, предоставящ услуга за мрежови печат с определен брой съоръжения за печат (сървъра вероятно работи с фиксиран "пул" от принтери, а не създава нови такива).

Ресурсът ще бъде моделиран от клас `Resource`, производителят и потребителите ще бъдат моделирани съответно от клас `Producer` и клас `ConsumerClient`.

Пример 2: Реализация на задачата производител-консуматор.

Consumer.java

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

class Consumer extends Thread {

    private Socket clientSocket;
    private Resource item;
    private Scanner input;
    private PrintWriter output;

    public Consumer( Socket clientSocket, Resource item ) {
        this.clientSocket = clientSocket;
        this.item = item;

        try {
            input = new Scanner(
                clientSocket.getInputStream() );
        }
        catch( IOException ioEx ) {
            System.out.println( "Error reading from socket." );
            System.out.println( "Closing down connection..." );
            clientSocket.close();
        }
    }

    @Override
    public void run() {
        String request = input.nextLine();

        if ( request.equals( "1" ) ) {
            item.takeOne();
            output.println( "Request granted." );
        }
    }

    public void shutdown() {
        try {
            System.out.println( "Closing down connection..." );
            clientSocket.close();
        }
        catch( IOException ioEx ) {
            System.out.println( "Unable to close connection to client!" );
        }
    }
}
```

```
        output = new PrintWriter(
            clientSocket.getOutputStream(), true );
    } catch( IOException ioEx ) {
        ioEx.printStackTrace();
    }
}

@Override
public void run() {

    String request = "";

    do {
        request = input.nextLine();

        if ( request.equals( "1" ) ) {
            item.takeOne();
            output.println( "Request granted." );
        }
    } while( !request.equals( "0" ) );

    try {
        System.out.println( "Closing down connection..." );
        clientSocket.close();
    } catch( IOException ioEx ) {
        System.out.println( "Unable to close connection to client!" );
    }
}
}
```

Producer.java

```
class Producer extends Thread {

    private Resource item;

    public Producer( Resource item ) {
        this.item = item;
    }

    @Override
    public void run() {
        int newLevel, pause;

        do {
            try {
                newLevel = item.addOne();
                pause = newLevel % 1000;
            }
            catch( IOException ioEx ) {
                System.out.println( "Error writing to socket." );
                System.out.println( "Closing down connection..." );
                clientSocket.close();
            }
        }
        catch( InterruptedException ie ) {
            System.out.println( "Thread interrupted." );
        }
    }
}
```

```

        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();
                output.println( message );
                response = input.nextLine();
                System.out.println( "SERVER> " +
                    response );
            if ( !message.equals( "0" ) &&
                !message.equals( "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 предстои да бъдат изпратени по даден канал или вече са прочетени от някои от каналите. В дадени ситуации е възможно дори реализиране на така наречените директни буфери. Директните буфери представляват ресурс на ниско ниво от самата операционна система. Java позволява извършване на операции на ниско,

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 единствен 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 своята универсалност. По своята същност буферът представлява масив. Размерът на този масив може да бъде зададен предварително:

```
buffer = ByteBuffer.allocate(2048);
```

Пример 3 е еквивалентното решение на разгледаният вече `MultiEchoServer`, чрез използване на мултиплексиране. Целта е да се сравнят изискванията към кода за варианти на многонишков сървър и сървър, изграден на принципа на NIO. Кодът за клиента не се налага да бъде представян, тъй като разликата в примерите се състои само в различните сървърни програми.

Пример 3: Неблокиращ сървър.

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class MultiEchoServerNIO {

    private static ServerSocketChannel serverSocketChannel;
    private static Selector selector;
    private static final int PORT = 1300;

    public static void main( String[] args ) {
        ServerSocket serverSocket;
        System.out.println( "Opening port...\\n" );

        try {
            serverSocketChannel =
                ServerSocketChannel.open();
            serverSocketChannel.configureBlocking(false );
            serverSocket = serverSocketChannel.socket();

            InetSocketAddress netAddress =

```

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

            selector = Selector.open();
            serverSocketChannel.register( selector,
                SelectionKey.OP_ACCEPT );
        } catch( IOException ioEx ) {
            ioEx.printStackTrace();
            System.exit( 1 );
        }

        processConnections();
    }

    private static void processConnections() {
        do {
            try {
                int numKeys = selector.select();
                if ( numKeys > 0 ) {
                    Set<SelectionKey> eventKeys =
                        selector.selectedKeys();
                    Iterator<SelectionKey> keyIterator =
                        eventKeys.iterator();

                    while( keyIterator.hasNext() ) {
                        SelectionKey key = keyIterator.next();
                        int keyOps = key.readyOps();
                        if ( ( keyOps & SelectionKey.OP_ACCEPT ) ==
                            SelectionKey.OP_ACCEPT ) {
                            acceptConnection( key );
                            continue;
                        }
                        if ( ( keyOps & SelectionKey.OP_READ ) ==
                            SelectionKey.OP_READ ) {
                            acceptData( key );
                        }
                    }
                }
            } catch( IOException ioEx ) {
                ioEx.printStackTrace();
                System.exit( 1 );
            }
        } while( true );
    }

    private static void acceptConnection(SelectionKey key) throws
IOException {
        SocketChannel socketChannel;
```

Bpmppcon n 3aa;aa;an:

1. Kofrro haqinha nma 3a cbs;ababe ha hñunra B Java? Karbn ihgejnicbra n he;jocatraru nmar te?
2. Hñamjherre upmepente ot ytpakheñnero n ahanjinspanate p3ejytrante.
3. Karbn ctpatreni sa peajin3auing ha ctpbrpn nma? Karbn ca texhite ihgejnicbra n he;jocatraru?
4. Peajin3anpabite ctpbrp sa ctpatinho ch;tpakheñe ihpe3 nojxoxa;c nmojorintra.

```

private static void accept(Socket socket) {
    try {
        selector.select();
        Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
        while (iterator.hasNext()) {
            SelectionKey key = iterator.next();
            if (key.isReadable()) {
                SocketChannel channel = (SocketChannel) key.channel();
                ByteBuffer buffer = channel.read(buffer);
                if (buffer.remaining() > 0) {
                    System.out.println("Received message from " + channel.getRemoteAddress());
                    String message = new String(buffer.array(), 0, buffer.remaining());
                    System.out.println("Message content: " + message);
                    buffer.clear();
                }
            } else if (key.isWritable()) {
                SocketChannel channel = (SocketChannel) key.channel();
                channel.write(buffer);
                if (buffer.remaining() == 0) {
                    channel.close();
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static void acceptData(SocketChannel channel, ByteBuffer buffer) {
    try {
        channel.read(buffer);
        if (buffer.remaining() == 0) {
            channel.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static void close(SocketChannel channel) {
    try {
        channel.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static void close(SocketChannel channel, ByteBuffer buffer) {
    try {
        channel.read(buffer);
        channel.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

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
Host IP address: 23.205.244.210
a23-205-244-210.deploy.static.akamaittechnologies.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 осъществяване на комуникация. Например:

```
Socket link = serverSocket.accept();
```

3. Задаване на входни и изходни потоци.

Методи *getInputStream* и *getOutputStream* на класа *Socket* се използват за получаване на референции към потоци, свързани с обекта от тип *Socket*, получен в стъпка 2. Тези потоци ще се използват за комуникация с клиента, създад текущата връзка. За програми, които нямат графичен потребителски интерфейс (GUI), може да се използва *Scanner* обект като обвивка (wrapper class) около *InputStream* обект, върнат от метода *getInputStream()*, за да се получи символно-ориентиран вход. Например:

```
Scanner input = new Scanner(link.getInputStream());
```

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 посредством метода *close()* на класа *Socket*. Например:

```
link.close();
```

Пример 2:

В този пример, сървърът ще получава съобщения от клиента под формата на символен низ и ще връща отговор с текущото клиентско съобщение с номер, под който е получено от сървъра. При получаване на съобщението *CLOSE* от страна на клиента комуникацията между клиент и сървър ще бъде преустановена. Тъй като по време на комуникацията при изпълнението на сокет процесите може да настъпи входно-изходна грешка (*IOException*), е необходимо използването на един или повече “try” блока в кода на програмата. Не е добра практика използването на един голям блок за прихващане на грешки, поради невъзможността за правилното

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

Добра практика изисква 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;
    private static final int PORT = 1230;

    public static void main( String[] args ) {
        System.out.println( "\nOpening port..." );
        try {
            socket = new ServerSocket( PORT ); // стъпка 1
        } catch ( IOException e ) {
            System.out.println( "\nUnable to attach port..." );
            System.exit( 1 );
        }

        do {
            handle();
        } while( true );
    }

    private static void handle() {
        Socket link = null; // стъпка 2
        Scanner input = null;
        try {
            link = socket.accept(); // стъпка 2
            // стъпка 3
            input = new Scanner( link.getInputStream() );
            PrintWriter output =
                new PrintWriter( link.getOutputStream(), true );
            int numMessages = 0;
```

```
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 close..." );
        System.exit( 1 );
    }
}
```

TCPEchoClient.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 TCPEchoClient {

    private static InetAddress host;
    private static int PORT = 1230;

    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() {
    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. Извличане 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. Извличане 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 пренатоварената форма на конструкторът му, която използва четири аргумента:

- байт масив, съдържащ съобщението отговор;
- размер на отговора;
- адрес на клиента;
- номер на порта на клиента.

Първият от тези аргументи се връща от метода `getBytes()` на String класа. Например:

```
DatagramPacket outPacket =
new DatagramPacket(response.getBytes(),
response.length(),clientAddress, clientPort);
```

където `response` (отговора) е променлива от тип String, съдържаща съобщението за отговор.

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 двета 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 изключение, генерирано от метода `close()`, което прави излишен единият от „try“ блоковете.

UDPEchoServer.java

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

public class UDPEchoServer {
    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 ) {
        System.out.println( "Opening port...\\n" );
    }
}
```

```

try {
    datagramSocket = new DatagramSocket( PORT );
} catch ( SocketException e ) {
    System.out.println( "Unable to open port..." );
    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( !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 разглеждаме, 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 вътрешният 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 или Java DataBase Connectivity е индустриски стандарт за данново-независима връзка на Java приложения към широк кръг SQL (релационни) бази от данни. Той осигурява три неща:

- установяване на връзка с базата от данни;
- изпращане на SQL заявки;
- обработка на резултатите.

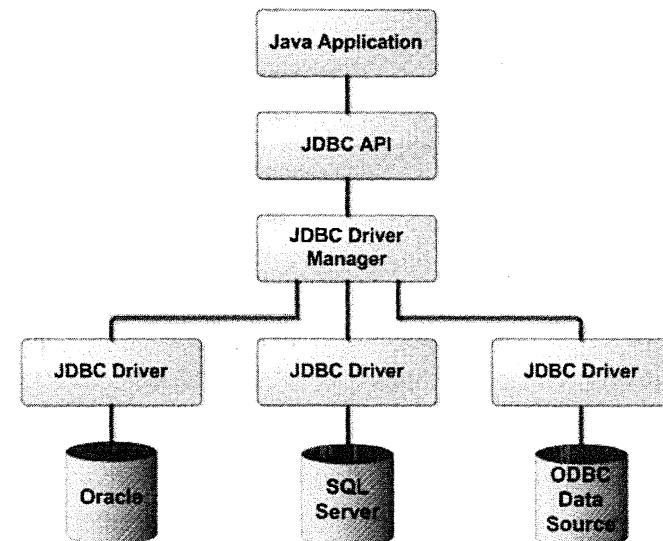
Това се осъществява чрез използване на драйвери за съответната СУБД. Различни Java технологии като Java приложения, Java аплети, Java Servlets, Java ServerPages (JSP), Enterprise JavaBeans (EJB) са в състояние да използват JDBC драйвер за достъп до база от данни

4.2 Архитектура на JDBC.

JDBC поддържа както двуслоен, така и трислоен модел за достъп до база от данни. Като цяло обаче JDBC архитектурата се състои от два слоя:

- JDBC API - осигурява връзката на приложението към JDBC мениджъра;
- JDBC драйвер интерфейс - осигурява връзката на JDBC мениджъра със съответния драйвер за базата от данни.

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:



Фиг.4.1 Архитектура на JDBC.

Основните пакети на JDBC са *java.sql* и *javax.sql*. Текуща актуална версия е JDBC 4.0. JDBC предоставя следните интерфейси и класове:

- DriverManager - управлява списъка с драйвери за бази от данни. Свързва заявките от Java приложението с правилния драйвер чрез използване на комуникационен subprotocol (Първият драйвер, който разпознае определен subprotocol, ще бъде използван за установяване на връзката с базата от данни);

- 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 съдържа;
- SQLException: Този клас съдържа всички грешки, които е възможно да се получат при комуникацията с базата от данни.

4.3 Система за управление на бази от данни.

Идеята за релационните бази от данни е описана за първи път от Dr. Edgar F. Codd в една научна публикация на IBM през 1970 година. Едва през 1977 година обаче излиза първата в света релационна база от данни, продукт на фирмата Software Development Laboratories под името Oracle V.2.

Понятието база от данни е нарицателно за системи, които съхраняват в себе си информация и обслужват заявки за достъп до тази информация. Такива системи се състоят от два основни слоя: слой на ниско ниво (на ниво файла система) и слой от високо ниво, осигуряващ изпълнението на бизнес логиката на цялата система (Система за Управление на База от Данни (СУБД))

4.3.1 MySQL.

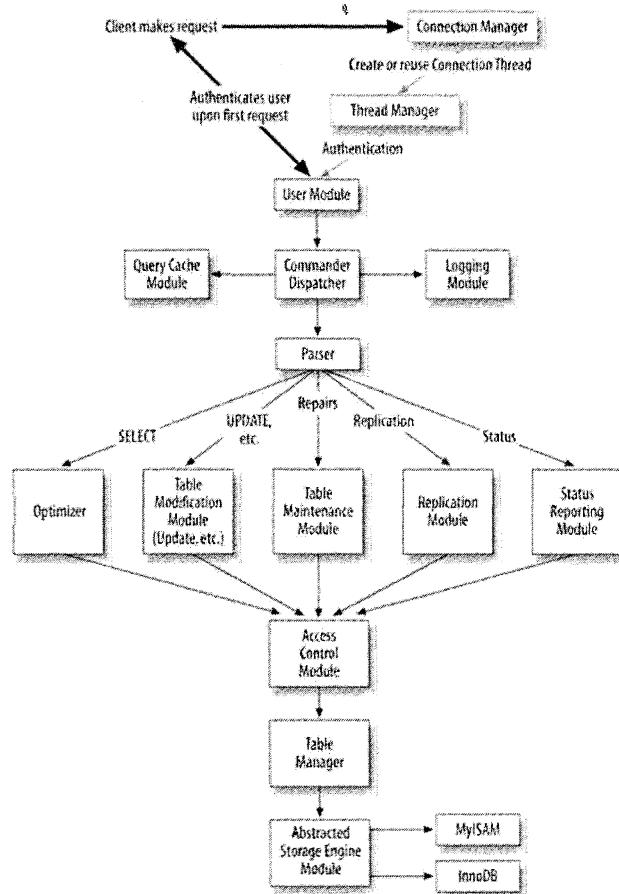
MySQL е една от най-популярните Open Source SQL системи за управление на бази от данни, която се поддържа от Oracle Corporation. MySQL е придобил такава популярност поради различни причини:

- реализиран е под open-source лиценз и е напълно безплатен;
- поддържа големия набор от функционалности и инструменти на останалите скъпи и мощни 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 GPL лиценза позволява модифициране на софтуера му за отговаряне на дадени специфични нужди.

MySQL се състои от много на брой модули, които имат за цел да организират процесите от гледна точка на комуникация с външния за базата от данни свят и управление на процесите и задачите по правилното съхранение на данните в самата база.

Има следните модули: Инициализиращ модул; Модул за управление на връзките; Модул за управление на нишките; Главна нишка за връзки; Модул за идентификация на достъпа; Модул за контрол на достъпа; Парсър; Диспечер на команди; Кеш на заявките; Модул за оптимизация; Модул за управление на таблиците; Абстрактен слой за съхранение на данни; Технология за съхранение на данните (Storage Engine) - MyISAM, InnoDB, MEMORY, Berkeley DB); и др.

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



Фиг.4.2 Архитектура на MySQL.

Когато СУБД се стартира, инициализацият модул поема контрола. Той прочита параметрите от конфигурационния файл и от командния ред (ако има такива), заделя памет за буферите, инициализира глобални променливи за структурите, зарежда таблицата за достъп до системата и други инициализиращи процеси.

След като процесите по инициализацията приключват, инициализацият 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 Данни 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 от -2147483648 до 2147483647. Ако е без знак, то диапазона става от 0 до 4294967295;
- **TINYINT** – стандартна променлива от целочислен тип (integer) за много малки числа, която може да е със или без знак. Ако е със знак, то диапазона и ще е от -128 до 127. Ако е без знак, то диапазона става от 0 до 255;

- **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 избирани. Лист "A" или "B" или

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

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 директория файлът mysqld с параметър *console* – *mysqld --console*.

```
C:\>cd mysql-5.6.21-winx64
C:\mysql-5.6.21-winx64>cd bin
C:\mysql-5.6.21-winx64\bin>mysqld --console
```

Фиг.4.3 Стартiranе на MySQL.

Ако всичко по инсталирането е наред, следва да се види еcran с различни системни мета-данни, в края на които е съобщението, че сървърът е стартиран успешно и слуша за връзки към него на порт 3306. Това е стандартният порт, на който работи MySQL.

Следващата стъпка има два варианта: или използване на конзолния интерфейс на MySQL и въвеждане на команди от команден ред (Фиг.4.4), до който се достига след изпълнението пак от директория /bin на файла *mysql*, или чрез използване на MySQL Workbench (Фиг.4.5).

```
C:\mysql-5.6.21-winx64>cd bin
C:\mysql-5.6.21-winx64\bin>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.6.21 MySQL Community Server (GPL)

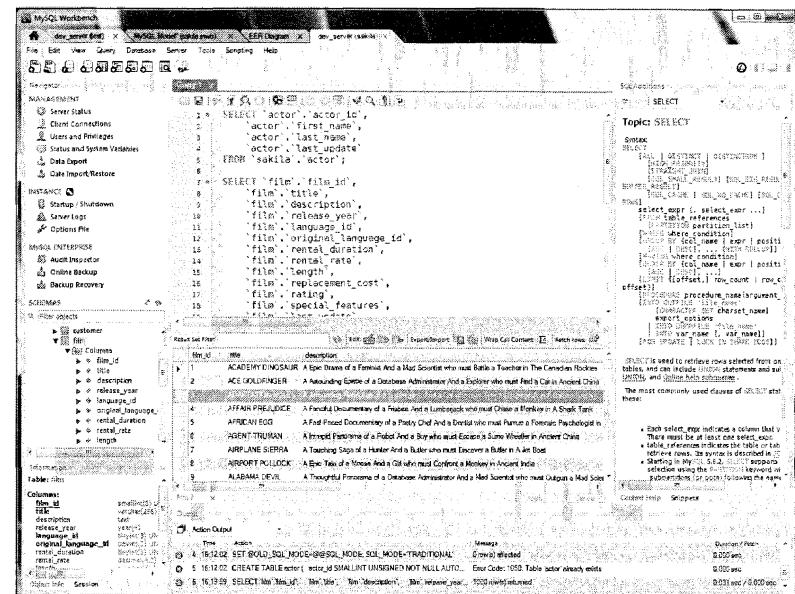
Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql>
```

Фиг.4.4 Конзолен интерфейс на MySQL.

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.



Фиг.4.5 MySQL Workbench.

4.4 Комуникация с база от данни.

Комуникацията с бази от данни в Java протича по следните стъпки:

- Създаване на връзка към базата;
- Изпълнение на заявки;
- Обработка на получените данни;
- Затваряне на връзката към базата от данни.

4.4.1 Създаване на връзка към базата.

Създаването на връзка се състои от следните етапи:

- 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
```

За да може въобще да бъде осъществена връзка с дадена СУБД, то трябва да бъде зареден съответстващият и драйвер от пакета с драйвери на JDBC. Съществуват различни драйвери в зависимост от компанията, поддържаща конкретната СУБД. В таблица 4.1 са представени най-често използваните СУБД.

Таблица 1

MySQL	com.mysql.jdbc.Driver
ORACLE	oracle.jdbc.driver.OracleDriver
DB2	COM.ibm.db2.jdbc.net.DB2Driver
Sybase	com.sybase.jdbc.SybDriver

4.4.1.2 Регистрация на драйвер.

За използване на драйвер за комуникация с MySQL СУБД, трябва да бъде свален от техния сайт съответно пакета необходим за това - `Connector/J` (<http://www.mysql.com/products/connector/>) . Мястото, където трябва да се постави, е директория `/lib` на проекта и съответно да се добави в `"build"` пътя на Java приложението на Java приложението.

Регистрацията на драйвер става по следните начини:

1. чрез `Class.forName()`:

```
try {
    Class.forName("com.mysql.jdbc.Driver");
}
```

```
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!");
    System.exit(2);
}
catchInstantiationException ex) {
    System.out.println("Error: unable to instantiate driver!");
    System.exit(3);
}
```

Използването на тези два метода макар да изглежда аналогично, си има своята съществена разлика. При първият подход `Class.forName()` се използва метод, чрез който може да бъде създаден пул от конекции към базата от данни. При вторият подход се използва Java *Singleton* дизайн шаблон, който гарантира, че ще бъде създадена една единствена връзка и всяко приложение, което иска да комуникира с базата от данни, ще използва точно тази връзка. С други думи, за да се свързват с базата от данни приложениета трябва да споделят една и съща връзка. Като продуктивност, първият подход е много по-добър, докато при втория се залага на сигурност от определена гледна точка.

Изборът за използване на кой да е от двата метода е според спецификата на използване на базата от данни. Защо? На пръв поглед използването на пул от връзки е изключително продуктивно и при натоварен множествен достъп до базата от данни използването на вече създадени конекции е изключително ефективно (създаването на TCP връзка е базен процес). От друга

страна, при не толкова динамично използване на връзките към базата, създаването и „замразяването“ на конекции е неефективно поради причина, че операционните системи имат таймаут за неизползвани TCP връзки, след изтичането на който те унищожават връзката.

3. чрез `DriverManager.registerDriver()`:

Регистрация на драйвер се използва когато се работи с Java виртуална машина (JVM) различна от тази на Oracle, например като такава, предоставена от Microsoft.

```
try {
    Driver myDriver = new com.mysql.jdbc.Driver();
    DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
    System.out.println("Error: unable to load driver class!");
    System.exit(1);
}
```

С въвеждането на JDBC 4.0 отпада необходимостта от регистрация на драйвер от програмиста чрез експlicitно използване на `Class.forName()`. `DriverManager` сам намира подходящия драйвер при извикване на метода `getConnection()`.

4.4.1.3 Задаване URL на базата.

Задаването на URL на базата е необходимо, за да може по-късно да се подаде като аргумент на метода за създаване на връзка. За различните СУБД форматът на задаване на адреса е различен. В таблица 4.2 са представени примери на най-често използваните:

Таблица 4.2

MySQL	<code>jdbc:mysql://hostname/ databaseName</code>
ORACLE	<code>jdbc:oracle:thin:@hostname:port</code> <code>Number:databaseName</code>
DB2	<code>jdbc:db2:hostname:port Number/databaseName</code>
Sybase	<code>jdbc:sybase:Tds:hostname: port</code> <code>Number/databaseName</code>

За комуникация с MySQL трябва да се регистрира следното URL:

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

4.4.1.4 Създаване на обект от тип `Connection`.

За създаването на обект от тип се използва метода `DriverManager.getConnection()`. Методът има три форми, които се използват за тази цел:

- `getConnection(String url)`
- `getConnection(String url, Properties prop)`
- `getConnection(String url, String user, String password)`

При първият метод потребителското име и парола се вграждат в URL на базата от данни:

```
try {
    ...
    ...
    ...
    connect = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/MyDataBase?"
        + "user=sqouser&password=squserpw");
```

При вторият метод се използва обект от тип `Properties`, в който се съхраняват потребителското име и парола и се подават като аргумент на метода:

```
import java.util.*;
...
String URL = "jdbc:mysql://localhost:3306/MyDataBase";
Properties info = new Properties();
info.put( "user", "username" );
info.put( "password", "password" );
...
Connection conn = DriverManager.getConnection(URL, info);
```

При третият случай URL, потребителско име и парола се подават като аргументи на метода:

```
String URL = "jdbc:mysql://localhost:3306/MyDataBase";
String USER = "username";
String PASS = "password";
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

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 параметри. Прекомпилирана заявка означава, че за разлика от обикновения *Statement* обект, който първо трябва да се компилира от MySQL и след това да се изпълни, тук това не се налага, а *PreparedStatement* е готов за директно изпълнение от СУБД. Този тип е особено ефективен при изпълнението на единотипна заявка с различни параметри;
- *CallableStatement* (Extends *PreparedStatement*) – използва се за достъпване и изпълнение на съхранени процедури в СУБД. Има както входни, така и изходни параметри.

Методите, които изпълняват *Statement* обектите, от своя страна също са три вида:

- *execute(String SQL)* – връща *true* ако първият обект, който се *ResultSet*. Този метод е подходящ за използване ако заявката може да върне повече от един *ResultSet* обект. В този случай различните обекти се достъпват като последователно се извиква методът *Statement.getResultSet*;
- *executeQuery(String SQL)* – връща само един обект от тип *ResultSet*. Подходящ за използване при SELECT заявка;
- *executeUpdate(String SQL)* – връща броя на редовете (като *int*), върху които изпълнението на заявката е оказало ефект.

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 извличане на данни** – извличане на данните от текущия ред, към който сочи курсора;
- **За актуализиране на данни** – актуализиране на данни от текущия ред, към който сочи курсорът. В последствие тази актуализация може да се приложи и върху базата от данни.

4.4.4 Затваряне на връзката с базата.

В края на изпълнението на заявките и комуникацията с базата е необходимо експлицитно да се затворят всички налични конекции към базата от данни. Ако това не се направи, по някое време garbage collector-а на Java ще го направи когато почиства ненужните обекти.

Позоваването на garbage collector-а е изключително лоша практика, особено в програмирането, свързано с бази от данни.

*Задължително е използването на метод *close()* за съответния обект, асоцииран с дадена връзка. За да е сигурно изпълнението на затварящият метод, той се поставя във „finally“ блок.*

Statement обект се затваря със *Statement.close*, което веднага освобождава ресурса, използван от обекта:

```
} finally {
    if (stmt != null) { stmt.close(); }
}
```

Обект от тип *Connection* се затваря също с метод *close()*:

```
conn.close();
```

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);

            System.out.println("Creating database...");
            stmt = conn.createStatement();

            String sql = "CREATE DATABASE STUDENTS";
            stmt.executeUpdate(sql);
            System.out.println("Database created successfully...");
        }catch(SQLException se){
            se.printStackTrace();
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            try{
                if(stmt!=null)
                    stmt.close();
            }catch(SQLException se2){
            }
            try{

```

```
                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...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            System.out.println("Connected database successfully...");

            System.out.println("Creating table in given database...");
            stmt = conn.createStatement();

            String sql = "CREATE TABLE MAGISTRI " +
                        "(id INTEGER not NULL, " +
                        " first VARCHAR(255), " +
                        " last VARCHAR(255), " +
                        " age INTEGER, " +
                        " PRIMARY KEY ( id ))";

            stmt.executeUpdate(sql);
            System.out.println("Created table in given database...");
        }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 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");

            System.out.println("Connecting to a selected database...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            System.out.println("Connected database successfully...");

            System.out.println("Inserting records into the table...");
            stmt = conn.createStatement();

            String sql = "INSERT INTO MAGISTRI " +
                "VALUES (100, 'Borislav', 'Dimitrov', 18)";
            stmt.executeUpdate(sql);
            sql = "INSERT INTO MAGISTRI " +
                "VALUES (101, 'Petar', 'Petrov', 25)";
            stmt.executeUpdate(sql);
            sql = "INSERT INTO MAGISTRI " +
                "VALUES (102, 'Todor', 'Todorov', 30)";

```

```

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 {

    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 = "SELECT id, first, last, age FROM MAGISTRI ";

```

```

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: Изтриране 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...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            System.out.println("Connected database successfully...");
            System.out.println("Creating statement...");
            stmt = conn.createStatement();
            String sql = "DELETE FROM MAGISTRI "
                    + "WHERE id = 101";
            stmt.executeUpdate(sql);
            sql = "SELECT id, first, last, age FROM Registration";
            ResultSet rs = stmt.executeQuery(sql);

            while(rs.next()){
                //Retrieve by column name
                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 7: Изтриране 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...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            System.out.println("Connected database successfully...");
            System.out.println("Deleting table in given database...");
            stmt = conn.createStatement();
            String sql = "DROP TABLE MAGISTRI ";
            stmt.executeUpdate(sql);
            System.out.println("Table deleted in given database...");
        }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 8: Изтриване на базата 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 = "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("Deleting database...");
            stmt = conn.createStatement();

            String sql = "DROP DATABASE STUDENTS";
            stmt.executeUpdate(sql);
            System.out.println("Database deleted successfully...");
        }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!");
}
}

```

Въпроси и задачи:

1. Какво представлява JDBC?
2. Какво място заема и каква роля има JDBC при изграждане на трислойна информационна система?
3. Какви подходи има при използването на JDBC за връзка с бази от данни?
4. Инсталирайте MySQL и JDBC конектора.
5. Изпълнете примерите от упражнението и анализирайте резултатите.
6. Направете база от данни „студент“, която да отразява данните на членовете на студентска група.

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 програми, наречени в този контекст клиентски софтуер (*client*), като обслужва техните заявки, подадени към него;
- Компютър, върху който се изпълнява сървърен софтуер, предоставящ една или повече услуги на други компютри в същата мрежа или Интернет пространството. В повечето случаи хардуерните изисквания към този компютър са по-високи от изискванията към хардуера на стандартния настолен компютър, който не функционира като сървър.

Сървърният софтуер се характеризира с това, че работи на принципа приложение, което слуша за заявки (*request*) и връща отговор (*response*), като и двете операции са по предварително зададен протокол. Тъй като заявката може да пристигне по всяко време, сървърният софтуер е в режим на непрекъснато изпълнение. Понятието „сървър“ е пряко свързано с модела за обслужване „клиент-сървър“, който представлява антипод на модела „peer to peer“.

Ако комуникацията между клиента и сървъра се осъществява с помощта на Hypertext Transfer Protocol (HTTP), а информацията, предавана като отговор към клиента, са HTML документи, които могат да включват изображения, стилове и скриптове в допълнение към съдържанието на текста, тогава говорим за уеб сървър (Web Server). При този тип сървъри заявките от клиентско приложение (web browser) се обслужват от уеб приложение, работещо на сървъра. Основните характеристики на този тип сървъри са следните:

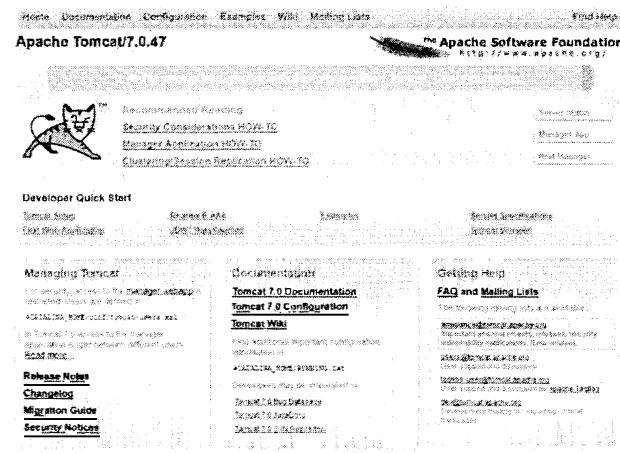
- наличие на съответствие между заявено от клиент URL и:
 - ✓ ресурс от локалната файлова система (за статични заявки);
 - ✓ локална или отдалечена програма (за динамични заявки);

- 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 предотвратяване претоварването на мрежата с цел увеличаване броя на клиентските заявки, които да се обработват;
- осигуряване на управление и сигурност за изпълняващите се на него уеб приложения.

Типични представители на този род сървъри са Apache HTTP Server на Apache Software Foundation, IBM HTTP Server на IBM, IIS на Microsoft, Jetty на Eclipse Foundation и др.

5.2 Apache Tomcat.

Apache Tomcat (често се споменава само като Tomcat), е уеб сървър и сървлет контейнер с отворен код, разработен от фондация Apache Software (ASF). Tomcat прилага няколко спецификации от Java EE (Java ентиърпрайз), включително Java Servlet, JavaServer Pages (JSP), Java EL и WebSocket, и осигурява "чиста Java" HTTP уеб сървър среда за изпълнение на Java код.



Фиг.5.1 Начална страница на Apache Tomcat сървър.

Tomcat е изграден на модулен принцип. Основните модули, на които се базира неговата работа, са следните:

Catalina

сървлет контейнер на Tomcat. Catalina изпълнява спецификациите на Sun Microsystems за сървлет и JavaServer Pages (JSP).

Coyote

компонент Connector за Tomcat, който поддържа HTTP 1.1 протокол като уеб сървър. Това позволява Catalina, който по принцип е Java Servlet или JSP контейнер, да работи и като обикновен уеб сървър, който предлага локални файлове като HTTP документи.

Jasper

Jasper е JSP Engine на Tomcat. Jasper прави разбор на JSP файлове, за да ги събере в Java код под формата на сървлети (които могат да се обработват от Catalina). По време на работа Jasper открива промени в JSP файлове и ги прекомпилира.

Tomcat 7.x (най-използваната към момента версия) изпълнява Servlet 3.0 и JSP 2.2 спецификации. Това изиска Java версия 1.6 и нагоре.

5.3 Java уеб приложения.

Концепцията на уеб приложение се появява с въвеждането на Java сървлет спецификация версия 2.2. Според тази спецификация, "уеб приложение е колекция от сървлети, HTML страници, класове, както и други ресурси, които могат да бъдат обединени и работят на множество контейнери от различни „доставчици“. Смисълът, който стои зад това е, че уеб приложението е контейнер, който може да съдържа всяка комбинация от следния списък обекти:

- Java Server Pages;
- Помощни класове (Utility classes);
- Статични документи, включително HTML, изображения, JavaScript; библиотеки, CSS стилове и т.н.;
- Класове за клиента;
- Мета-информация, описваща уеб приложението.

Съдържание, което може да се използва на всеки сървър, съвместим с Java EE спецификацията.

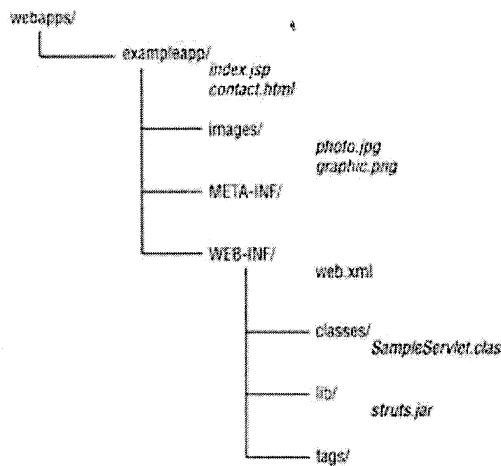
Едно добре проектирано уеб приложение става преносимо между повечето уеб сървъри, без да се изискват промени в неговото съдържание. Всяко уеб приложение има собствена сесия, контекст, клас за зареждане на контекста (*loader*) и структура на директориите. Тази конструкция гарантира достатъчна степен на координация между различните компоненти на приложението, като същевременно позволява отделяне от другите приложения, които могат да се изпълняват на един и същ уеб сървър.

Структура на web приложение

Едно от предимствата на уеб приложениета е, че те отговарят на стандарт за разположението на различните си типове ресурси. За да работи правилно уеб сървъра, всеки един от уеб компонентите трябва да е разположени на съответното му място. По-долу е описано местоположението на повечето стандартни части на уеб приложение:

- /WEB-INF/lib директория - JAR файлове (съдържа пакети от Java класове, осигуряваща определена функционалност);
- /WEB-INF - web.xml, описател, който осигурява основната конфигурация за уеб приложението;
- /WEB-INF/classes – директория за самостоятелни Java класове.

Този стандарт за организиране на приложението спомага за намаляването на времето за стартиране на нови приложения, както и улеснява процеса му на развой от страна на програмистите. Фигурата по-долу показва стандартните места за ресурси в едно примерно уеб приложение, използващо Tomcat уеб сървър:



Фиг.5.2 Примерна структура на уеб приложение.

От фиг.5.2 се вижда, че уеб страниците (независимо дали статичен HTML или динамичен JSP или някакво съдържание от друг шаблонен език за генериране на динамично съдържание) се разполагат в главната директория на приложението или нейни поддиректории, но не в /WEB-INF или /META-INF директорийните структури. /WEB-INF директорията има няколко специфики в своето съдържание. В поддиректорията си /classes тя съдържа всички необходими за приложението java класове, които не са пакетирани в някакъв JAR файл. В поддиректория /lib се поставят всички необходими за приложението JAR пакети от класове. В /WEB-INF се поставя ако ще се използва и описателят за инсталлиране на приложението (deployment descriptor) – файлът web.xml. Той съдържа конфигурации на приложението, описание на уеб ресурсите и други допълнителни настройки.

Когато различни уеб приложения са разположени на един и същ уеб сървър, Java Servlet Technology модела е проектиран по такъв начин, че уеб приложения не си пречат едно на друго. Всяко приложение има свой собствен унифициран указател на ресурс

(URL), който се използва за достъп до него и има собствен *ServletContext* обект, за да общува с уеб сървъра. От ключово значение е да се отбележи че при изпълнението на дадено уеб приложение, когато уеб сървъра получи заявка от клиент, не уеб приложението е нещото, което обработва заявката, а уеб контейнера, в който то е разположено. Контейнерът след това предава на уеб приложението изпълнението като извика необходимия метод от него. Това е един от основните начини, по който уеб приложенията се съхраняват разделени по време на тяхното изпълнение на един и същ уеб сървър. Резултатът, който се получава, е че уеб приложенията работят все едно са разположени на отделни сървъри, тъй като в по-голямата част от случаите те не трябва да знаят едно за друго.

*Когато е необходимо уеб приложенията да комуникират едно с друго, има стандартен начин за осъществяването на тази комуникация (обикновено чрез *ServletContext*).*

Една от основните характеристики на уеб приложението е неговата връзка със *ServletContext*. Всяко приложение си има свой собствен *ServletContext*. Тази връзка се управлява от сървлет контейнера и гарантира, че две приложения няма да изпаднат в конфликт при достъп на обекти от *ServletContext*.

Инсталиране (внедряване) на приложения (Deploy)

Приложенията, които следват да се инсталат на сървъра, се подават към него под формата на специален вид архив. Архивите, предназначени за тази роля, са няколко вида и те следват като структура и съдържание стандартите и спецификациите на Java EE:

- Web Application Archive (WAR) – състои се от уеб компоненти като сървлети и JSP страници, както и статични HTML файлове, JAR файлове, тагови библиотеки и инструментални класове и др. WAR файлът има разширение .war;
- EJB JAR - съдържа ентърпрайз бийнове и компоненти, необходими за EJB технологията. Включва също всякакви

необходими инструментални класове. EJB JAR файлът има разширение *.jar*.

- J2EE Application Client JAR - съдържа код за клиент на J2EE приложение, което има достъп до сървърни компоненти (напр. ентърпрайз бийнове) чрез RMI/JNDI протокол. Името на файла има разширение *.jar*.
- Resource Adapter Archive (RAR) - съдържа адаптер за даден ресурс. Определя се от спецификациите на J2EE Connector Architecture, представлява преносим компонент, който дава възможност на ентърпрайз бийн компоненти, уеб компоненти, както и приложения на клиенти да имат достъп до ресурси. Разширението на архива е *.rar*.
- Enterprise Application Archive (EAR) - съдържа един или повече WAR, EJB JAR или RAR файлове. Разширението на архива е *.ear*.

Едно приложение може да бъде асемблирано в един EAR файл или в отделни WAR, EJB JAR, и клиент JAR файлове. От гледна точка на административните средства, разполагането на обектите и командите за внедряване на приложениета са сходни за всички видове файлове.

Конвенции за именуване

Модули от различни видове могат да имат едно и също име в рамките на едно и също приложение. Когато приложението се разполага на сървъра, директориите, съдържащи отделните модули, са именувани с *_jar*, *_war* и *_ear* надставки. Модули от същия вид в рамките на програма трябва да имат уникални имена. Например имената на файловете на схема за база данни трябва да са уникални в рамките на приложението.

С цел избягване на колизии при именуването на обекти е препоръчително да се използва схемата за задаване на имена, дефинирана в спецификацията на Java EE. Установяване на конвенция за именуване и нейното строго спазване улеснява много цялостния процес по създаване и внедряване на приложения.

Въпроси и задачи:

1. Инсталирайте Apache Tomcat Server.
2. Разгледайте и анализирайте директорийната структура на сървъра.
3. Стартирайте сървъра и разгледайте готовите примери, достъпни чрез */localhost*.
4. Как се организира контрола над потребителите в Tomcat?
5. Как се инсталира *web* приложение в Tomcat?
6. Как може да се промени използванятия по подразбиране от сървъра порт 80? Каква е възможната нова стойност?

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, бази от данни или друг софтуер посредством сокети и RMI механизми.

Като обобщение може да се каже, че сървлетите предоставят компонентно-базиран, платформено-независим метод за изграждане на уеб-базирани приложения, без ограниченията на CGI програмите. Сървлетите имат достъп до цялото семейство от Java програмните интерфейси (API), включително API JDBC за достъп до бази от данни.

Сървлетите изпълняват следните основни задачи:

- четене на данни, изпратени от клиенти (браузъри). Това включва HTML форми на уеб страница, аплет или някакъв специфичен HTTP клиент, бисквитки (cookies), видове медии, схеми за компресиране и така нататък;
- да обработва данните и да генерира резултати. Този процес може да изисква комуникация с база данни, изпълнение на

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 извика метода си *service()*, за да обслужва клиентските заявки;
- сървлетът е деактивиран чрез повикване на метода му *destroy()* от контейнера;
- сървлетът е премахнат от паметта от почистващата услуга на джава виртуалната машина (JVM).

Методът init()

Методът е проектиран да бъде викан само веднъж. Извиква се при създаването на сървлета и повече не се вика. Използва се за първоначална инициализация. Сървлетът обикновено се създава когато клиентът за първи път достъпи URL-то, на което той съответства, но може да бъде създаден и при стартирането на сървъра. Когато клиент изпрати заявка за достъп до сървлет се създава единична негова инстанция, а обслужването на всяка клиентска заявка се обработва в отделна нишка. Методът *init()* също така създава или зарежда различни помощни данни, необходими за работата на сървлета. Дефиницията на метода е следната:

```
public void init() throws ServletException{
    // инициализираш 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 сървлърът 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
}
```

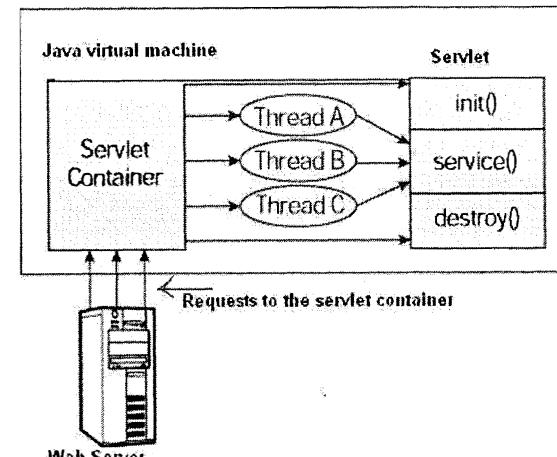
Методът *destroy()*

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(){
    // финализираш 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
    {
    }
}
```



Hello Java

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>

```

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

```

@WebServlet("/HelloForm")
public class HelloForm extends HttpServlet {
    public void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Using GET 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><b>First Name</b>: "
        + request.getParameter("first_name") + "\n"
        + "<li><b>Last Name</b>: "
        + request.getParameter("last_name")
        + "\n" + "</ul>\n" + "</body></html>");
    }
}

```

Обработване на POST заявка, HTML форма:

```

<html>
<body>

    <form action="HelloForm" method="POST">

```

```

        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>

```

*Сървъл, 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><b>First Name</b>: "
        + request.getParameter("first_name") + "\n"
        + " <li><b>Last Name</b>: "
        + request.getParameter("last_name")
        + "\n" + "</ul>\n" + "</body></html>");
    }
}

```

Въпроси и задачи:

1. Какви са начините за създаване на Сървъл?
2. До коя част от жизнения цикъл на сървълата има достъп програмиста?
3. По какъв начин сървълите обработват клиентските заявки?
4. Изпълнете примерите и анализирайте получния резултат.
5. Как може да се контролира достъпа до *doGet* и *doPost* методите на сървълата?
6. Създайте комуникация между форма и сървъл с и без използване на *web.xml*.

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 представява 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 регистрация на предпочитанията 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 включвания, 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

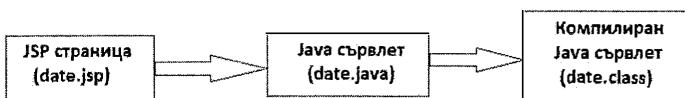
сървър, за да изпълнява сложни задачи като достъп до бази от данни и обработка на изображения и т.н.

- 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.
- JSP енджинът зарежда JSP файла от диска и го конвертира в съдържание на сървлет. Това представлява преобразуване на всички текстове от структурата на файла в `println()`, а всички JSP елементи се превръщат в Java код. JSP енджинът компилира сървлета в изпълним клас и изпраща първоначалната клиентска заявка към сървлет-енджина;
- Сървлет-енджинът зарежда получния Servlet клас-файл и го изпълнява. По време на изпълнение сървлета генерира съдържание в HTML формат, което сървлет-енджина предава към уеб сървъра като HTTP отговор;
- Уеб сървърът изпраща на браузъра отговора под формата на статично HTML съдържание;
- Накрая уеб браузъра обработва динамично генерираната HTML страница.



Фиг.8.1 Фази на JSP страницата.

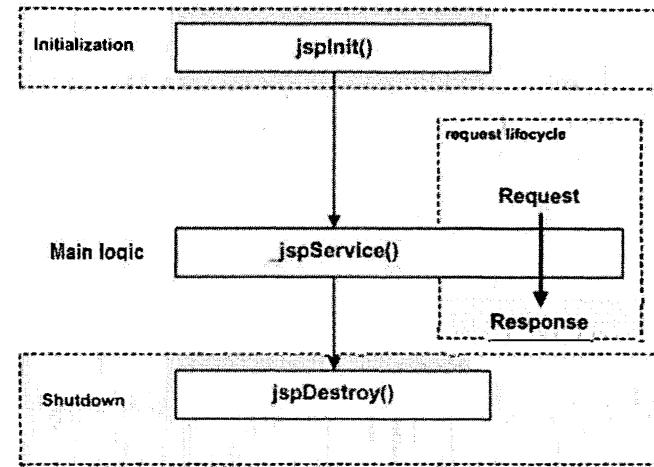
Обикновено JSP енджина проверява дали вече съществува сървлет, съответстващ на текущото JSP, и дали датата на изменението на JSP страницата е по-стара от сървлета. Ако тя е по-стара, контейнерът приема, че JSP страницата не е променяна и че генерирания за нея сървлет все още отговаря на съдържанието ѝ.

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 трансляция 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 създаването на страницата до унищожаването ѝ; процес, който е подобен на жизнения цикъл на сървлета с допълнителна стъпка трансформацията от JSP в сървлет.



Фиг.7.2 Жизнен цикъл на JSP.

Етапите, през които минава JSP страницата, са следните:

- Компиляция;
- Инициализация;
- Изпълнение;
- Почистване (изтриване).

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 елемент 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 декларират една или повече променливи или методи, които да са достъпни за използване по-късно във файла JSP. Синтаксисът на декларацията е следния:

```
<%! declaration; [ declaration; ]+ ... %>
```

Пример:

```
<html>
<body>
<%! int data = 50; %>
<%= "Value of the variable is:" + data %>
</body>
</html>
```

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:

```
<%@ directive attribute="value" %>
```

Има три типа директиви:

- **<%@ page ... %>** - дефинира инструкции към контейнера, които се отнасят за текущата JSP ;
- **<%@ include ... %>** - включване на файл в процеса на трансформация от JSP в сървлет;
- **<%@ taglib ... %>** - за декларация на библиотека с дефинирани тагове от потребителя, използвани на конкретната страница.

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 сървлет 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 сървлет-енджина. Позволяват събития като например динамично вмъкване на файл, повторно използване на JavaBeans компоненти, препращане на потребителя към друга страница, или генериране на HTML за Java плъгина. Синтаксисът на елемента за действие е следният:

```
<jsp:action_name attribute = "value" />
```

Елементите за действие са основно предварително дефинирани функции и съществуват следните JSP действия:

- *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 плъгин;
- *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 уеб 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>
    <%
      if (day == 1 || day == 7) {
    %>
    <p>Today is weekend</p>
    <%
      } else {
    %>
    <p>Today is not weekend</p>
    <%
```

```
}
```

```
%>  
</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 />  
        <%  
          fontSize++;  
        %>  
        <%  
          %>  
      }  
    <%  
    %>  
  </body>  
</html>
```

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