Интернет програмиране с Java

Неофициален работен вариант на учебника по едноименната дисциплина

Автор: Светлин Наков,
Преподавател по едноименната дисциплина в
Софийски Университет "Св. Климент Охридски",
Факултет по Математика и Информатика
Web-site: http://www.nakov.com

Съавтор на част от темите: Борис Червенков,
e-mail: boris@abv.bg

Web-site на курса: http://inetjava.sourceforge.net

Последна промяна: 25.11.2002

Курсът "Интернет програмиране с Java" е предназначен за всички, които се интересуват от програмиране на Java и разработка на Интернет-ориентирани приложения и има за цел да запознае читателя със следните технологии:

- Socket програмиране – разработка на Java приложения, които комуникират по Интернет/Интранет по протоколите TCP/IP, например Chat клиент/сървъри, Web-сървъри, Mail клиент/сървъри и др.

- Java аплети – разработка на малки Java приложения, които могат да се вграждат във Web страници и да се изпълняват от Web-браузъра на клиента.

- Web-приложения – разработка на Web приложения с технологиите Servlets и Java Server Pages (JSP), създаване и deploy-ване на Web-приложения съгласно стандартите на Sun за J2EE, работа със сървъра Tomcat.

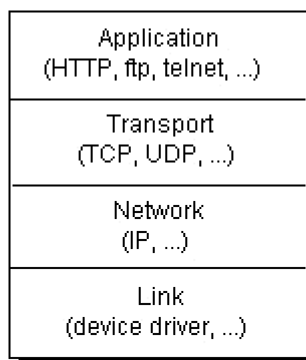
За да бъде разбран материала, е необходимо читателите да имат основни познания по организация на Интернет, програмиране, Java и HTML. Поради големият си обем, темата ще бъде разделена на няколко части (уроци), които ще бъдат публикувани в няколко последователни броя на списанието. За настоящия курс е създаден и сайт в Интернет на адрес: http://inetjava.sourceforge.net, където има много допълнителна информация по темата (на български и английски език), както и дискусионен форум.

Основи на Интернет – адреси, портове, протоколи, клиент/сървър приложения

Не можем да започнем един практически курс по разработка на Интернет приложения, без да засегнем, поне частично, основните принципи на които се основава пренасянето на данни в световната мрежа. В тази тема ще разгледаме накратко най-важните неща, които имат пряко отношение към мрежовото програмиране с Java. Предполагаме, че читателят има основни познания по организация на Интернет и затова няма да обясняваме подробно някои общоизвестни термини като например Web-сървър, "протокол" и т.н.

Основно понятие в Интернет е ***IP адрес***. Това е уникален 32-битов номер на компютър, който се записва като четири 8-битови числа, разделени по между си с точки. За улеснение на потребителите някои компютри в Интернет освен IP адрес могат да имат и име. Съответствията между IP адресите и имената на компютрите се поддържат от специални DNS сървъри, които могат да намират IP адрес по име на компютър и обратното.

Комуникацията между компютрите в Интернет е организирана на няколко нива, както показва следната диаграма:



Когато пишем Java програми които комуникират по мрежата, ние програмираме най-горния слой от диаграмата, така нареченият Application слой. Преносът на данни, предизвикан от нашите Java програми, обикновено се осъществява от транспортния слой посредством протоколите TCP или UDP. Транспортният слой използва по-долния мрежов слой за прехвърляне на малки количества информация, наречени IP пакети, от един компютър на друг, а тези пакети се прехвърлят чрез мрежови протоколи и връзки на още по-ниски нива. Като програмисти на Java, не е необходимо да знаем в детайли за всичко това, но все пак трябва да имаме представа поне от TCP и UDP протоколите дотолкова, доколкото е необходимо да преценим кога кой от тях да използваме и от IP протокола дотолкова, доколкото е необходимо да знаем, че всеки компютър в Интернет си има IP адрес, по който можем да се обръщаме към него.

TCP е протокол, който осигурява надежден двупосочен комуникационен канал между две приложения. Можем да сравним този канал с канала, по който се осъществява обикновен телефонен разговор. Например, ако искаме да се обадим на приятел, ние набираме неговия номер и когато той вдигне, се осъществява връзка между нас двамата. Използвайки тази връзка, ние можем да изпращаме и получаваме данни от нашия приятел, до момента, в който един от двамата затвори телефона и прекрати връзката. Подобно на телефонните линии, TCP протоколът гарантира, че данните, изпратени от едната страна на линията, ще се получат от другата страна на линията без изменение и то в същия ред, в който са изпратени. Ако това е невъзможно по някаква причина, ще възникне грешка и ние ще разберем, че има някакъв проблем с комуникационния канал. Именно заради тази своя надеждност, TCP е най-често използваният протокол за трансфер на информация по Интернет. Примери за приложения, които комуникират по TCP са Web-браузърите, Web-сървърите, FTP клиентите и сървърите, Mail клиентите и сървърите – приложения, за които редът на изпращане и пристигане на данните е много важен.

UDP е протокол, който позволява изпращане и получаване на малки независими един от друг пакети с данни, наречени дейтаграми, от един компютър на друг. За разлика от TCP, UDP не гарантира нито реда на пристигане на изпратените последователно дейтаграми, нито гарантира, че те ще пристигнат въобще. Изпращането на дейтаграма е като изпращане на обикновено писмо по пощата: редът на пристигане на писмата не е важен и всяко писмо е независимо от останалите. UDP се използва значително по-рядко от TCP заради това, че не осигурява комуникационен канал за данни, а позволява изпращане само на единични независими кратки пакети.

Както TCP, така и UDP протоколът позволява едновременно да се осъществяват няколко независими връзки между два компютъра. Например можем да зареждаме няколко различни Web-сайта чрез нашия Web-браузър и същевременно да теглим през FTP няколко различни файла от един или няколко FTP сървъра. Реално погледнато едно и също приложение (например нашият Web-браузър) отваря едновременно няколко независими комуникационни канала до един или няколко различни сървъра, като по всеки от тях прехвърля някаква информация. За да е възможно няколко приложения да комуникират по мрежата едновременно, е необходимо пакетите информация, предназначени за всяко от тях да бъдат обработени от него, а не от някой друг, за да може всяко приложение да получи своите данни. Именно за решаване на този конфликт се използват портовете в протоколите TCP и UDP. ***Портът*** е число между 0 и 65536 и задава идентификатор на връзката в рамките на машината. Всеки TCP или UDP пакет съдържа в себе си

освен данните, които пренася, още 4 полета, описващи от кого до кого е изпратен пакета: source IP, source port, destination IP и destination port. По IP адресите се разпознават компютрите, отговорни за изпращане и получаване на съответните пакети, а по портовете се разпознават съответните приложения, работещи на тези компютри, които изпращат или трябва да получат информацията от тези пакети. Всяка TCP връзка в даден момент се определя еднозначно от порт и IP източник и порт и IP получател. Комуникационният канал, който предоставя една TCP връзка наричаме ***сокет***. Например нека нашият IP адрес е 212.50.1.81, и сме стартирали Internet Explorer и Outlook Express. С Internet Explorer ние браузваме някакъв сайт и за целта той е отворил няколко сокета към IP адрес 212.50.1.1 на порт 80 и тегли някакви Web-страници и картинки. В същото време с Outlook Express си теглим пощата, и за целта той е отворил сокет към 192.92.129.4 на порт 110. В този момент имаме няколко едновременно отворени TCP сокета (няколко независими една от друга комуникационни линии), чрез които нашият компютър комуникира с други два компютъра. Можем да ги представим схематично по следния начин:

Internet Explorer = 212.50.1.81:1033 ↔ 212.50.1.1:80 = Apache Web Server
Internet Explorer = 212.50.1.81:1037 ↔ 212.50.1.1:80 = Apache Web Server
Outlook Express = 212.50.1.81:1042 ↔192.92.129.4:110 = Microsoft Exchange POP3 Server

Първата връзка служи за изтегляне на някаква Web-страница. Тя има за източник приложението Internet Explorer и за нея е определен порт източник 1033 на нашия компютър (212.50.1.81). За получател е определен компютърът 212.50.1.1 и порт получател 80, който порт е свързан с приложението, което обслужва достъпа до Web-страниците на този компютър – Apache Web Server. Източник и получател не е съвсем точно казано, защото всички TCP връзки са двупосочни, т.е. предоставят два независими канала за данни за всяка от посоките, но все пак можем да приемем за източник това приложение, което е създало връзката (отворило сокета). Втората връзка служи за изтегляне на някаква картинка и прилича много на първата, но с една разлика – портът източник. Този порт източник е свързан също с приложението Internet Explorer на нашия компютър, но е друго число. Въпреки, че двете връзки са между едни и същи приложения, те са различни и независими, т.е. представляват два независими канала за данни. Единия служи за изтегляне на някакъв HTML документ, а другият за изтегляне на някаква картинка. Web-сървърът знае по кой от двата канала да изпрати HTML документа и по кой картинката. Internet Explorer също знае по кой от двата канала ще пристигне HTML документа и по кой картинката. Това се определя от порта източник, който е различен за двата канала. Портът източник се задава автоматично от операционната система при създаване на TCP сокет. Този порт е уникален в рамките на машината. Портът получател се задава от програмиста заедно с IP адреса получател за определяне на компютъра и приложението, към което програмата иска да се свърже. Програмистът трябва да знае предварително IP адреса и порта на приложението, с който иска да осъществи комуникация.

Съществуват два вида приложения, които комуникират по TCP протокола – клиентски и сървърски. ***Клиентските приложения*** се свързват към сървърските като отварят сокет към тях. За целта те предварително знаят техните IP адреси и портове. ***Сървърските приложения*** "слушат на определен порт" и чакат клиентско приложение да се свърже към тях. При пристигане на заявка за връзка от някой клиент на порта, на който сървърът слуша, се създава сокет за връзка между клиента на неговия порт източник и сървъра на неговия порт получател. Забележете, че клиентите отварят сокети към сървърите, а сървърите създават сокети само по клиентска заявка, т.е. те не отварят сокети. Едно клиент/сървър приложение можем да си представим като магазин с няколко щанда и клиенти, които пазаруват в него. Сървърът може да се сравни с магазин, а портът, на който слуша този сървър – с определен щанд вътре в магазина. Когато дойде клиентът, той се допуска, само ако иска да отиде на някой от щандовете, които работят (допуска се връзка само на отворен порт /порт на който слуша някое сървърско приложение/). Когато клиентът отиде на съответния щанд, той започва да си говори с продавача (осъществява комуникационна линия и прехвърля данни по нея в двете посоки) на определен език, който и двамата разбират (предварително известен протокол за комуникация). Както магазинът, така и щандът могат да обслужват няколко клиента едновременно, без да си пречат един на друг. След приключване на комуникацията клиентът си тръгва (и затваря сокета). Междувременно продавачът може да изгони клиента от магазина, ако той се държи невъзпитано или няма пари (сървърът може да затвори

сокета по всяко време). За повечето операции със сокети имаме аналог с нашия пример с магазина и затова взаимодействието клиент/сървър лесно може да се интерпретира като взаимодействие потребител на услуга/извършител на услуга.

Третата връзка от показаните по-горе свърза приложението Outlook Express, което се идентифицира с порт 1042 на нашата машина (212.50.1.181) с приложението Microsoft Exchange POP3 Server, което се идентифицира с порт 110 на машината с IP адрес 192.92.129.4. Пристигналите TCP пакети на нашата машина ще бъдат разпознати от операционната система по четирите полета, които идентифицират един сокет – source IP, source port, destination IP и destination port и ако са валидни, информацията от тях ще се предаде на съответното приложение. Понеже едно приложение, както видяхме може да отвори повече от един сокет до някое друго приложение, е най-правилно да се каже, че портът източник и портът получател задават не само клиентското и сървърското приложение съответно, но и идентификатора на връзката в рамките на тези приложения (или по-точно в рамките на машината).

При UDP комуникацията концепцията с портовете е същата, само че не се осъществява комуникационен канал между приложенията, а се изпращат и получават отделни единични пакети. Тези пакети носят в себе си същата допълнителна информация като TCP връзките – IP и порт на изпращач и IP и порт на получател. И при UDP протокола също има клиентски и сървърски приложения и по същият начин операционната система разпознава кой пакет за кое приложение е.

Комуникационните канали, наречени сокети не са достатъчни за осъществяване на комуникация. Подобно на ситуацията в магазина, клиентът трябва да комуникира със сървъра на някакъв предварително известен и за двамата език – ***протокол***. В Интернет съществува голямо количество стандартни протоколи за комуникация между приложенията, като всеки от тях е свързан с някаква услуга. Всяка услуга работи с някакъв протокол. Например услугата достъп за Web-ресурс работи по протокола HTTP, услугата за изпращане на e-mail работи по протокола SMTP, а услугата за достъп до файл от FTP сървър работи по протокола FTP. За всяка от тези стандартни Интернет услуги има и асоциирани стандартни номера на портове, на които тези услуги се предлагат. Стандартните портове са въведени за да се улесни създаването на клиентски приложения, понеже всяко клиентско приложение трябва да знае не само IP адреса или името на сървъра, на който се предлага услугата, до която то иска достъп, но също и порта, на който тази услуга е достъпна. Ето и няколко стандартни портове, протоколи и услуги:

порт	протокол	услуга
21	FTP	Услуга за достъп до отдалечени файлове. Използва се от FTP клиенти (например Internet Explorer, GetRight, CuteFTP)
25	SMTP	Услуга за изпращане на E-mail. Използва се от E-mail клиенти (например Outlook Express)
80	HTTP	Услуга за достъп до Web-ресурси. Използва се от Web-браузъри (например Internet Explorer)
110	POP3	Услуга за извличане на E-mail от пощенска кутия. Използва се от E-mail клиенти (например Outlook Express)

Java програмите могат да използват TCP и UDP протоколите за комуникация през Интернет чрез класовете от пакета `java.net`. По късно ще бъдат разгледани в детайли съответните класове, с който се могат да се създават TCP и UDP клиент/сървър приложения – `InetAddress`, `Socket`, `ServerSocket`, `DatagramSocket`, `DatagramPacket`, `URL` и др., но преди това ще направим кратък преглед на средствата за вход/изход и многонишково програмиране в Java, защото те са важна основа за разбиране на по-нататъшния материал.

Вход/изход в Java – кратък преглед

В тази тема ще направим съвсем кратък преглед на най-важните класове и методи за вход и изход в Java. Всичко останало може да се намери с документацията на JDK 1.3.

В езика Java входно-изходните операции са базирани на работа с потоци от данни. ***Потоците*** са канали за данни, при които достъпът се осъществява само последователно.

Класовете, чрез които се осъществяват входно-изходните операции се намират в пакета `java.io`. Има два основни типа потоци – текстови и бинарни.

Текстовите потоци служат за четене и писане на текстова информация, а бинарните – за четене и писане на двоична информация. Базов за всички входни текстови потоци е интерфейсът `java.io.Reader`, а за всички изходни текстови потоци – `java.io.Writer`. Най-удобен за четене от текстови потоци е класът `java.io.BufferedReader`, който предлага метод за четене на цял текстов ред `readLine()`. За писане в текстови потоци е най-удобен класът `java.io.PrintWriter`, който има метод `println(…)`. Ето един прост пример за използване на текстови потоци – програма, която номерира редовете на текстов файл:

```
import java.io.*;
import java.lang.*;

public class TextFileLineNumberInserter {
   public static void main(String[] args) throws Exception {
       FileReader inFile = new FileReader("input.txt");
       BufferedReader in = new BufferedReader(inFile);

       FileWriter outFile = new FileWriter("output.txt");
       PrintWriter out = new PrintWriter(outFile);

       int counter = 0;
       String line;
       while ( (line=in.readLine()) != null ) {
           counter++;
           out.println(counter + ' ' + line);
       }

       in.close();
       out.close();
   }
}
```

Трябва да отбележим, че въпреки че Java работи вътрешно с Unicode стрингове, текстовите потоци четат и пишат символите не в Unicode, а като използват стандартните 8-бита за символ. При писане и четене информацията се преобразува от и към Unicode по текущо-активната кодова таблица, което създава известни проблеми. Това е една от причините, заради която не можем да обработваме бинарна информация с текстови потоци.

Базов за всички входни двоични потоци е интерфейсът `java.io.InputStream`, а за всички изходни двоични потоци е интерфейсът `java.io.OutputStream`. Ключов метод на `InputStream` е методът `read(byte[] b)`, който чете данни от входния поток и ги записва в масив, а ключови методи в OutputStream са `write(byte[] b, int off, int len)`, който изпраща данни от масив към изходния поток и `flush()`, който изпразва буферите и записва чакащата в тях информация.

Важно е да се съобразяваме с факта, че записването на данни, както в текстов, така и в двоичен изходен поток не винаги предизвиква тяхното изпращане. Ето защо по-нататък в нашата работа когато изпращаме двоични или текстови данни по сокет, винаги ще викаме метода `flush()`, за да сме сигурни, че данните са отпътували по сокета, а не чакат в някой буфер. Ето и фрагмент от програмен код, който копира двоични файлове:

```
FileInputStream inFile = new FileInputStream("input.bin");
FileOutputStream outFile = new FileOutputStream("output.bin");
byte buf[] = new byte[1024];
while (true) {
   int bytesRead = inFile.read(buf);
   if (bytesRead <= 0) break;
   outFile.write(buf, 0, bytesRead);
}
outFile.close();
inFile.close();
```

Многонишково програмиране в Java

В тази тема ще се запознаем с възможностите за многонишково програмиране с Java, тъй тези знания ще са ни крайно необходими в по-нататъшната ни работа.

Многонишковите (***multithreaded***) програми представляват програми, които могат да изпълняват едновременно няколко редици от програмни инструкции. Всяка такава редица от програмни инструкции наричаме ***thread*** (нишка). Изпълнението на многонишкова програма много прилича на изпълнение на няколко програми едновременно. Например в Microsoft Windows е възможно едновременно да слушаме музика, да теглим файлове от Интернет и да въвеждаме текст. Тези три действия се изпълняват от три различни програми (процеси), които работят едновременно. Когато няколко процеса в една операционна система работят едновременно, това се нарича многозадачност. Когато няколко отделни нишки в рамките на една програма работят едновременно, това се нарича ***multithreading*** (многонишковост). Например ако пишем програма, която работи като Web-сървър и Mail-сървър едновременно, то тази програма трябва да изпълнява едновременно поне 3 независими нишки – една за обслужване на Web заявките (по протокол HTTP), друга за изпращане на поща (по протокол SMTP) и трета за теглене на поща (по протокол POP3). Освен това е много вероятно за всеки потребител на тази програма да се създава по още една нишка, за да може този потребител да се обслужва независимо и да не бъде каран да чака, докато системата обслужва останалите.

С Java създаването на многонишкови програми е изключително лесно. Достатъчно е да наследим класа `java.lang.Thread` и да припокрием метода `run()`, в който да напишем програмния код на нашата нишка. След това можем да създаваме обекти от нашия клас и с извикване на метода им `start()` да започваме паралелно изпълнение на написания в тях програмен код. Ето един пример:

```
class MyThread extends Thread {
   private String name;
   private long timeInterval;

   public MyThread(String name, long timeInterval) {
       this.name = name;
       this.timeInterval = timeInterval;
   }

   public void run() {
        try {
           while (!isInterrupted()) {
               System.out.println(name);
               sleep(timeInterval);
           }
       } catch (InterruptedException intEx) {
       }
   }
}

public class ThreadTest
{
   public static void main(String[] args) {
       MyThread thread1 = new MyThread("thread 1", 1000);
       MyThread thread2 = new MyThread("thread 2", 2000);
       MyThread thread3 = new MyThread("thread 3", 1500);
       thread1.start();
       thread2.start();
       thread3.start();
   }
}
```

След стартиране на тази програма се създават и стартират 3 нишки от класа `MyThread`. Всяка от тях в безкраен цикъл печата на конзолата името си и изчаква някакво предварително зададено време между 1 и 2 секунди. Понеже трите нишки работят паралелно, се получава резултат подобен на този:

```
thread 1
thread 2
thread 3
thread 1
thread 3
thread 2
thread 1
thread 3
```

```
thread 1
...
```

Трябва да отбележим, че прекратяването на thread не трябва да става насилствено чрез метода `stop()`, а thread-ът трябва учтиво да бъде помолен да прекрати работата си чрез извикване на `interrupt()`. Затова всеки thread в програмния си код трябва да проверява, извиквайки метода `isInterrupted()`, дали не е помолен да прекрати работата си. Други интересни методи на класа `Thread` са `setPriority(int)`, `sleep()` и `setDaemon()` и за тях можем да прочетем в документацията. Средствата за синхронизация на thread-ове са изнесени в класа `java.lang.Object` в методите `wait()` и `notify()`, но за тях ще говорим в следващата тема.

Синхронизация на нишки в Java

В предходната тема изяснихме какво е нишка (***thread***) и как се разработват многонишкови приложения с Java. В тази тема ще се запознаем с възможностите за синхронизация при многонишковото програмиране.

Има много ситуации, в които няколко нишки едновременно осъществяват достъп до общ ресурс. Например в една банка може едновременно двама клиенти да поискат да внесат пари по една и съща сметка. Да преположим, че сметките са обекти от класа Account, а операциите върху тях се извършват от класа Bank:

```
public class Account {
    private double mAmmount = 0;

    void setAmmount(double aAmmount) {
        mAmmount = aAmmount;
    }

    double getAmmount() {
        return mAmmount;
    }
}

public class Bank {
    public static void deposit(
        Account aAcc, double aSum)
    {
        double oldAmmount = aAcc.getAmmount();
        double newAmmount = oldAmmount + aSum;
        aAcc.setAmmount(newAmmount);
    }
}
```

Нека двамата клиенти се опитат едновременно да внесат съответно 100 и 500 лева в сметката `acc`, която е празна. Това би могло да стане по следния начин:

Клиент 1: `Bank.deposit(acc, 100);`

Клиент 2: `Bank.deposit(acc, 500);`

Както се вижда от кода, алгоритъмът за внасяне на пари към сметка работи така: Прочита сумата от сметката. Добавя сумата за внасяне към нея. Записвата новата сума в сметката. Ако заявките за внасяне на пари от двамата клиента се изпълняват едновременно, ще се получи следният неприятен ефект: Клиент 1 прочита сумата от сметката – 0 лева. Клиент 2 прочита сумата от сметката – също 0 лева. Клиент 1 прибавя към прочетената сума 100 лева и записва в сметката новата сума – 100 лева. Клиент 2 прибавя към прочетената сума 500 лева и записва в сметката новата сума – 500 лева. В резултат в сметката се получават 500 вместо 600 лева, а това за една банка това е абсолютно недопустимо. Натъкнахме се на класически синхронизационен проблем.

Когато няколко конкуретни нишки или няколко отделни програми се опитват едновременно да осъществят достъп до общ ресурс, се получават неприятни ефекти подобни на този с банката. Такива ефекти наричаме синхрознизационни проблеми, а техниката за решаването им наричаме синхронизация.

Синхронизацията решава проблемите с конкурентния достъп до общи ресурси, като прави достъпа до тях последователен. Тя предизвиква подреждане на заявките в последователност по такъв начин, така че когато една заявка се изпълнява, всички останали я чакат и се изпълняват едва след като тя приключи. Този процес съвсем не е автоматичен и се задава от програмиста чрез средствата за синхронизация, които ни дава операционната система или платформата, за която разработваме софтуер.

В Java средствата за синхронизация са вградени. Запазената дума `synchronized` предизвиква синхронизирано изпълнение на програмен блок. Това означава, че две нишки не могат едновременно да изпълняват програмен код този блок. Ако едната е започнала изпълнение на код от блока, другата ще я изчака да завърши. Проблемът с банката можем да решим много просто, като заменим декларацията на метода `deposit()` от горната програма

```
public static void deposit(
    Account aAccount, double aSum)
```

с декларацията

```
synchronized public static void deposit(
    Account aAccount, double aSum).
```

Запазената дума `synchronized`, зададена при декларацията на метод предизвиква синхронизиране на изпълнението на този метод по обекта, на който той принадлежи, а при статични методи – по класа, на който той принадлежи. Синхронизацията на програмен код по някакъв обект предизвиква заключване на този обект при започване на изпълнението на синхронизирания код и отключване при завършване на изпълнението му. Когато някоя нишка се опита да изпълни синхронизиран код, чийто обект е заключен, тя принудително изчаква отключването на този обект. Така код, синхронизиран по един и същ обект, не може да се изпълнява от две нишки едновременно и заявките за изпълнението му се изпълняват една след друга. В Java синхронизацията може да става по всеки обект, защото е вградена в началния за цялата класова йерархия базов клас `java.lang.Object`. В горния пример чрез ключовата дума `synchronized` синхронизирахме достъпа до метода `deposit()` по банката, което означава, че двама клиенти не могат да бъдат едновременно обслужвани от нея. Въпреки, че това решава проблема, такъв подход не е правилен, защото заключва цялата банка, вместо само сметката, с която се работи. За да заключваме само сметката, до която методът `deposit()` осъществява достъп можем да използваме следния синхронизиран код:

```
public static void deposit(
    Account aAccount, double aSum)
{
    synchronized (aAccount) {
        double oldAmmount = aAccount.getAmmount();
        double newAmmount = oldAmmount + aSum;
        aAccount.setAmmount(newAmmount);
    }
}
```

Това вече решава правилно проблема, защото заключва само сметката, която се променя, а не цялата банка. Важно е когато се използва заключване на обекти, да се заключва само този обект, който се променя и то само за времето, през което се променя, а не за по-дълго, за да може конкурентните нишки да чакат минимално при опит за достъп до него. Освен това, ако достъпът до някакъв обект трябва да е синхронизиран, той трябва да е синхронизиран навсякъде, където се работи с този обект. В противен случай полза от синхронизацията няма. Например ако в нашата банка внасянето на пари е синхронизирано, а тегленето не е синхронизирано, възможността за грешки при финансовите операции ще си остане съвсем реална.

Въпреки че синхронизацията чрез запазената дума `synchronized` върши работа в повечето случаи, тя съвсем не е достатъчна. За това в класа `java.lang.Object` съществуват още няколко важни метода свързани със синхронизацията – `wait()`, `notify()` и `notifyAll()`. Методът `wait()` приспива текущата нишка по даден обект докато друга нишка не извика `notify()` за същия обект за да я събуди. Методът `notify()` събужда една от заспалите по даден обект нишки, а `notifyAll()` събужда всичките. Ако по обекта няма заспали нишки, `notify()` и `notifyAll()` не правят нищо.

Понякога за работата на една нишка е необходим ресурс, който се получава в резултат от работата на друга нишка. В този случай първата нишка трябва да изчака втората да свърши някаква работа и след това да продължи своето изпълнение. Ако първата нишка в един цикъл постоянно проверява дали втората е свършила очакваната работа, тя ще консумира по неразумен начин много процесорно време и ще намали производителността на цялата система. Много по-ефективно е първата нишка да заспи, очаквайки събуждане от втората, а втората да свърши очакваната работа и веднага след това да събуди първата. Характерното за заспалите (или както още се наричат блокирали) нишки е, че не консумират процесорно време, което е причината приспиването на нишки да бъде предпочитан начин за чакане на ресурси.

Да разгледаме един класически проблем, известен като "производител – потребител". Един завод произвежда някаква продукция и разполага със складове в които може да побере някакво определено количество от нея. Когато складовете се напълнят заводът спира работа докато не продаде част от продукцията за да освободи място. Търговците от време на време идват в складовете и купуват част от произведената продукция. Когато търговец дойде и складът е празен, той чака докато заводът произведе продукция, за да му я продаде. Взаимодействието между производителя (завода) и потребителите (търговците) представлява постоянен процес, в който всеки върши своята работа, но същевременно зависи от другите и ги изчаква ако е необходимо. Проблемът "производител – потребител" се изразява в това да се организира коректно многонишковият процес на взаимодействие между производителя и потребителите без да се отнема излишно процесорно време когато някой чака някого за някакъв ресурс. Нека ресурсите, които производителят произвежда и потребителите консумират са текстови съобщения, а буферът, с който производителят разполага, е опашка с вместимост 5 съобщения. Следната е програма е примерно решение на проблема, реализирано на базата на средствата за синхронизация в Java:

```
import java.util.*;

class Producer extends Thread {
    static final int MAXQUEUE = 5;
    private Vector messages = new Vector();

    public void run() {
        try {
            while (true) {
                putMessage();
                sleep(1000);
            }
        } catch(InterruptedException e) { }
    }

    private synchronized void putMessage()
        throws InterruptedException
    {
        while (messages.size() == MAXQUEUE)
            wait();
        messages.addElement(
            new Date().toString() );
        notify();
    }

    // Called by Consumer
    public synchronized String getMessage()
        throws InterruptedException
    {
        notify();
        while (messages.size() == 0)
            wait();
        String message =
            (String)messages.firstElement();
        messages.removeElement( message );
        return message;
    }
}

class Consumer extends Thread {
    Producer producer;
```

```
    Consumer(Producer p) {
        producer = p;
    }

    public void run() {
        try {
            while (true) {
                String message =
                    producer.getMessage();
                System.out.println(message);
                sleep(1500);
            }
        } catch( InterruptedException e ) { }
     }
}

public class TestProducerConsumer {
    public static void main(String args[]) {
        Producer producer = new Producer();
        producer.start();
        Consumer c1 = new Consumer(producer);
        c1.start();
        Consumer c2 = new Consumer(producer);
        c2.start();
    }
}
```

Разглеждайки кода на програмата и документацията на методите `wait()` и `notify()`, вероятно ще ви направи впечатление, че тези методи могат да се викат само от код, който е синхронизиран по обекта, на който те принадлежат. Ако методът за заспиване и методът за събуждане се викат от различни нишки и са в блокове код, синхронизирани по един и същ обект, би трябвало след заспиването на първата нишка кодът, който я събужда никога да не се изпълни, защото ще чака завършването на заспалата нишка. Изглежда, че и двете нишки ще заспят за вечни времена, като едната ще чака другата да я събуди, а другата ще чака първата да излезе от синхронизирания код, но това няма да се случи никога понеже е заспала. Тези разсъждения, обаче са погрешни, защото извикването на `wait()` не само приспива текущата нишка, но и отключва обекта, по който тя е синхронизирана. Това позволява на блокът извикващ `notify()`, който е синхронизиран по същия обект, да се изпълни. Извикването на `notify()` събужда заспалата нишка, но не й разрешава веднага да продължи изпълнението си. Събудената нишка изчаква завършването на синхронизирания блок, от който е извикан `notify()`. След това продължава изпълнението си като заключва отново синхронизационния обект и го отключва едва след завършването на синхронизирания блок, в който е била заспала. Така заспиването за вечни времена, наричано още deadlock или мъртва хватка не настъпва. При неправилна употреба на средствата за синхронизация, обаче, настъпването на мъртва хватка съвсем не е изключено. Отговорност на програмиста е да предотврати възможността две или повече нишки в някой момент да започнат да се чакат взаимно.

Работа с TCP сокети

Както вече знаем от краткия преглед на Интернет протоколите, който направихме в началото, TCP сокетите представляват надежден двупосочен транспортен канал за данни между две приложения. Приложенията, които си комуникират през сокет, могат да се изпълняват на един и същ компютър или на различни компютри, свързани по между си чрез Интернет или друга TCP/IP мрежа. Тези приложения биват два вида – сървъри и клиенти. Клиентите се свързват към сървърите по IP адрес и номер на порт чрез класа `java.net.Socket`. Сървърите приемат клиенти чрез класа `java.net.ServerSocket`. Да разгледаме основната част от кода на едно просто сървърско приложение – DateServer:

```
ServerSocket srvSock = new ServerSocket(2002);
while (true) {
    Socket socket = srvSock.accept();
    OutputStreamWriter out =
```

```
            new OutputStreamWriter(
                socket.getOutputStream() );
        out.write(new Date()+ "\n");
        out.close();
        socket.close();
    }
```

Този сървър отваря за слушане TCP порт 2002, след което в безкраен цикъл приема клиенти, изпраща им текущата дата и час и веднага след това затваря сокета с тях. Отварянето на сокет за слушане става като се създава обект от класа `ServerSocket`, в конструктура на който се задава номера на порта. Приемането на клиент се извършва от метода `accept()` на класа `ServerSocket`, при извикването на който текущата нишка блокира до пристигането на клиентска заявка, след което създава сокет между сървъра и пристигналия клиент. От създадения сокет сървърът взема изходния поток за изпращане на данни към клиента (чрез метода `getOutputStream()`) и изпраща в него текущата дата и час, записани на една текстова линия. Затварянето на изходния поток е важно. То предизвиква действителното изпращане на данните към клиента, понеже извиква метода `flush()` на изходния поток. Ако нито един от методите `close()` или `flush()` не бъде извикан, клиентът няма да получи нищо, защото изпратените данни ще останат в буфера на сокета и няма да отпътуват по него. Накрая, затварянето на сокета предизвиква прекъсване на комуникацията с клиента. Сървъра можем да изтестваме със стандартната програмка telnet, която е включена в повечето версии на Windows, Linux и Unix като напишем на конзолата следната команда:

```
telnet localhost 2002
```

Резултатът е получената от сървъра дата:

```
Sat May 18 22:46:55 EEST 2002
```

Нека сега напишем клиент за нашия сървър – програма, която се свързва към него, взема датата и часа, които той връща и ги отпечатва на конзолата. Основната част от кода е следната:

```
Socket socket = new Socket("localhost", 2002);
BufferedReader in = new BufferedReader(
    new InputStreamReader(
        socket.getInputStream() ) );
System.out.println(in.readLine());
socket.close();
```

Свързването към сървър става чрез създаването на обект от класа `java.net.Socket`, като в конструктора му се задават IP адреса или името на сървъра и номера на порта. От свързания успешно сокет се взема входния поток и се прочита това, което сървърът изпраща. След приключване на работа сокетът се затваря. При работа със сокети и входно-изходни потоци понякога възникват грешки, в резултат на което се хвърлят изключения (exceptions). Затова е задължително и в двете програми, които дадохме за пример, кодът, който комуникира по сокет да бъде поставен в `try ... catch` блок, за да бъдат обработени евентуално възникналите грешки. Грешки възникват в най-разнообразни ситуации. Например ако сървърът не е пуснат и клиентът се опита да се свърже с него, ако връзката между клиента и сървъра се прекъсне, ако сървърът се опита да слуша на зает вече порт и в много други случаи. Също от важно значение е, че ако клиентът се опита да прочете данни от сървъра, а той не му изпрати нищо, клиентът ще блокира до затваряне на сокета. Затова сървърът и клиентът трябва да комуникират по предварително известен и за двамата протокол и да го спазват стриктно.

Даденият по-горе пример за сървър обслужва клиентите последователно. Ако двама клиенти едновременно дадат заявка, първият ще бъде обслужен веднага, а вторият едва след приключване на обслужването на първия. Тази стратегия работи, но само за прости сървъри, в които обслужването на клиент отнема много малко време. В повечето случаи обслужването на един клиент отнема известно време и останалите клиенти не могат да бъдат карани да го чакат. Затова се налага сървърът да се обслужва клиентите едновременно и независимо един от друг. За реализация на тази стратегия в средата на Java се използва многонишковия подход, при който за всеки клиент се създава отделна нишка. Това е препоръчвания начин за разработка на сървъри, предназначени да работят с повече от един клиент. Ще демонстрираме силата на многонишковия подход с един стандартен пример – сървър за разговори (chat server):

```
import java.io.*;
import java.net.*;
import java.util.Vector;

public class NakovChatServer {
    public static void main(String[] args)
    throws IOException {
        ServerSocket serverSocket =
            new ServerSocket(5555);
        System.out.println("Nakov Chat Server"
            + " started on port " +
            serverSocket.getLocalPort());

        ServerMsgDispatcher dispatcher =
            new ServerMsgDispatcher();
        dispatcher.start();

        while (true) {
            Socket clientSock =
                serverSocket.accept();
            ClientListener clientListener =
                new ClientListener(clientSock,
                    dispatcher);
            dispatcher.addClient(clientSock);
            clientListener.start();
        }
    }
}

class ClientListener extends Thread {
    private Socket mSocket;
    private ServerMsgDispatcher mDispatcher;
    private BufferedReader mIn;

    public ClientListener(Socket aSocket,
    ServerMsgDispatcher aServerMsgDispatcher)
    throws IOException {
        mSocket = aSocket;
        mIn = new BufferedReader(
            new InputStreamReader(
                mSocket.getInputStream()));
        mDispatcher = aServerMsgDispatcher;
    }

    public void run() {
        try {
            while (!isInterrupted()) {
                String msg = mIn.readLine();
                if (msg == null) break;
                mDispatcher.dispatchMsg(
                    mSocket, msg);
            }
        } catch (IOException ioex) { }
        mDispatcher.deleteClient(mSocket);
    }
}

class ServerMsgDispatcher extends Thread {
    private Vector mClients = new Vector();
    private Vector mMsgQueue = new Vector();

    public synchronized void addClient(
    Socket aClientSocket) {
        mClients.add(aClientSocket);
    }

    public synchronized void deleteClient(
    Socket aClientSock) {
        int i = mClients.indexOf(aClientSock);
        if (i != -1) {
            mClients.removeElementAt(i);
            try {
                aClientSock.close();
```

```
            } catch (IOException ioe) {}
        }
    }

    public synchronized void dispatchMsg(
    Socket aSocket, String aMsg) {
        String IP = aSocket.getInetAddress().
            getHostAddress();
        String port = "" + aSocket.getPort();
        aMsg = IP + ":" + port + " : " +
            aMsg + "\n\r";
        mMsgQueue.add(aMsg);
        notify();
    }

    private synchronized String
    getNextMsgFromQueue()
    throws InterruptedException {
        while (mMsgQueue.size() == 0)
            wait();
        String msg = (String) mMsgQueue.get(0);
        mMsgQueue.removeElementAt(0);
        return msg;
    }

    private synchronized void
    sendMsgToAllClients(String aMsg) {
        for (int i=0; i<mClients.size(); i++) {
            Socket socket =
                (Socket) mClients.get(i);
            try {
                OutputStream out =
                    socket.getOutputStream();
                out.write(aMsg.getBytes());
                out.flush();
            } catch (IOException ioe) {
                deleteClient(socket);
            }
        }
    }

    public void run() {
        try {
            while (true) {
                String msg =
                    getNextMsgFromQueue();
                sendMsgToAllClients(msg);
            }
        } catch (InterruptedException ie) { }
    }
}
```

Като функционалност сървърът не е сложен. Единственото, което прави той е да приема съобщения от клиентите си и да изпраща всяко прието съобщение до всеки клиент, като отбелязва в него от кого го е получил. Можем да го изтестваме като отворим няколко telnet-сесии към порт 5555 по същия начин, както в предния пример с Date-сървъра.

Сървърът има две основни нишки. Едната е главната програма (`NakovChatServer`), която слуша на порт 5555 и приема нови клиенти, а другата е нишката-диспечер (`ServerMsgDispatcher`), която разпраща получените от клиентите съобщения до всички свързани към сървъра. За всеки клиент в сървъра се създава още една нишка (обект от класа `ClientListener`), която служи за получаване на съобщения от него. При стартирането си сървърът отваря за слушане порт 5555, създава диспечера за съобщения и го стартира. След това в безкраен цикъл започва да приема клиенти, да ги добавя в списъка на диспечера, да създава по една нишка за получаване на съобщенията от тях и да я стартира. Нишката за получаване на съобщения от клиент в основния си цикъл (метода `run()`) чете съобщения от клиента, добавя ги в опашката на диспечера (извиквайки метода `DispatchMsg()`), след което го събужда ако е заспал (като му `notify()` метода). Четенето на съобщение става с метода `readLine()` и е операция, която блокира нишката докато не пристигне съобщение или не настъпи грешка. При настъпване на грешка, клиентът се премахва от списъка на

диспечера (чрез извикване на `deleteClient()`). Нишката `ServerMsgDispatcher` е добър пример за използване на проблема “производител – потребител” в практиката. В основния си цикъл (в метода `run()`) нишката взема от опашката си поредното съобщение и го разпраща до всички клиенти. В този цикъл тя се явява консуматор на съобщения. Ако опашката е празна, нишката чака (извиквайки `wait()`) докато пристигне ново съобщение. Съобщенията пристигат асинхронно чрез извиквания от нишките за обслужване на клиент. Клиентите играят ролята на производител на съобщения. Диспечерът пази всички активни клиенти в един списък. За да поддържа списъка актуален, сървърът добавя в него всеки нов клиент при пристигането му и го премахва от там при първия неуспешен опит за изпращане или получаване на съобщение от него. Така например, ако клиентът затвори сокета, той ще бъде премахнат от списъка, защото четенето на съобщение от него ще се провали. Макар и сървърът да е способен да обслужва много клиенти едновременно, не може все още да твърди, че е добре написан. Проблемът е, че нишката-диспечер, която изпраща получените съобщения, работи последователно. Тя не преминава към изпращането на следващо съобщение от опашката докато не изпрати текущото на всички клиенти. Ако връзката с един от клиентите е много бавна, заради него всички ще се наложи да чакат преди да получат следващото изпратено съобщение. Ето защо е необходимо диспечерът да работи паралелно. Единият вариант е да се създава по една нишка за всяко изпращане на съобщение до някой клиент. Това обаче означава, че ако сървърът има 1000 клиента и получи почти едновременно 100 съобщения, ще трябва да създаде 100000 нишки, за да разпрати съобщенията до клиентите. Създаването и изпълнението на толкова много нишки обаче, изисква огромно количество процесорно време и памет (особено при програмиране на Java), така че ще ни е необходим доста мощен компютър за да може така модифицираният chat-сървър да работи със задоволителна скорост. Ето защо е по-разумно за всеки клиент да се създаде по една нишка, която служи за разпращане на съобщенията, предназначени за него. Тази нишка трябва да поддържа опашка от съобщения, защото ако съобщенията пристигат по-бързо отколкото се могат да се изпратят, ще възникне проблем. Тя трябва да заспива, когато опашкатае празна и да се събужда, когато в нея постъпи съобщение. Тази нишка може да се реализира по същия начин като класа `ServerMsgDispatcher`, защото има почти същата функционалност. Примерна реализация на този подход има на сайта на курса “Интернет програмиране с Java” – http://inetjava.sourceforge.net.

Работа с UDP сокети

В предходната тема изяснихме как се разработват Java приложения, които си комуникират чрез TCP сокети. Както знаем от първата част на настоящия курс, TCP е протокол, който осигурява надеждна двупосочна поточно-ориентирана комуникационна линия. В тази тема ще се занимаен със средствата, които платформата Java ни дава за комуникация чрез единични UDP пакети.

Както знаем, протоколът UDP осигурява изпращане и получаване на единични пакети с данни, пристигането на които не е гарантирано. Поради факта, че не установява връзка между двата компютъра, които комуникират по между си, UDP генерира по-малък мрежов трафик отколкото TCP и затова осигурява по-голяма бързина. Той може да се използва, когато трябва да се изпращат малки по размер и независими едно от друго съобщения, но не е подходящ за изпращане на съобщения с голям размер или ако редът на тяхното пристигане е важен. Освен това, както вече знаем, успешното изпращане на един UDP пакет не гарантира че той ще пристигне или пък че ако изпратим два UDP пакета, те ще пристигнат в същия ред, в който са изпратени. Ето защо преди да се вземе решение да се използва комуникация по UDP, трябва да се прецени дали този протокол е подходящ.

В Java за поддръжката на UDP сокети разполагаме с класовете `java.net.DatagramSocket` и `java.net.DatagramPacket`. Класът `DatagramSocket` ни дава възможност да се свърваме към определен мрежов интерфейс и порт, да изпращаме пакети и да получаваме пакети. Класът `DatagramPacket` представлява структура от данни, която описва един UDP пакет. Да илюстрираме използването на тези класове чрез един пример. Нека създадем приложението UDPDataServer, което отговаря на клиентски UDP заявки за взимане на текущата дата:

```
import java.net.*;
```

```
public class UDPDateServer {
    public static void main(String[] args)
    throws Exception {
        DatagramSocket socketIn =
            new DatagramSocket(12345);
        DatagramSocket socketOut =
            new DatagramSocket();
        System.out.println(
            "UDP Date Server started.");
        while (true) {
            // Receive a request from a client
            byte[] requestBuf = new byte[256];
            DatagramPacket packetIn =
                new DatagramPacket(requestBuf,
                    requestBuf.length);
            socketIn.receive(packetIn);
            int len = packetIn.getLength();
            String request =
                new String(requestBuf, 0, len);

            // Send a response to the client
            if (request.startsWith("PORT=")) {
                int port = new Integer(request.
                    substring(5)).intValue();
                String resp = new java.util.
                    Date().toString();
                byte[] respBuf =
                    resp.getBytes();
                DatagramPacket packetOut =
                    new DatagramPacket(respBuf,
                        respBuf.length,
                        packetIn.getAddress(),
                        port);
                socketOut.send(packetOut);
            }
        }
    }
}
```

Както се вижда от кода, сървърът отваря два UDP сокета – един на порт 12345 за получаване на UDP пакети и още един на случаен порт, през който да изпраща отговорите на клиентите. След това в безкраен цикъл получава клиентска заявка, като счита че тя не надвишва 256 байта, извлича от нея IP адреса и порта, където да изпрати отговор, създава отговор (символен низ, съдържащ текущата дата и час) и го изпраща на клиента. IP адресът на клиента се взима от адреса, от който е дошъл пакета със заявката, а портът се взима от текста на клиентската заявка, която трябва да е във формат "`PORT=число`". Важна особеност на програмирането с UDP сокети с Java е, че един обект от класа `DatagramSocket` може да се използва или само за получаване, или само за изпращане на UDP пакети. Ако един сървър получава пакет, а след това изпраща отговор, той трябва да създаде два сокета, с които да извършва това, както е в нашия пример.

Нека сега напишем клиент за нашия сървър. Той трябва да изпраща на сървъра заявка, в която да задава на кой порт да се получи отговора и след като получи този отговор, да го отпечата. Разбира се, получаването на отговор не е гарантирано и затова чакането му трябва да е ограничено откъм време. Често срещана грешка е да се опитваме да получим отговора на същия порт, от който сме изпратили заявката. На пръв поглед това изглежда разумно, защото няма да има нужда да изпращаме порта, на който да получаваме отговора. Сървърът би могъл да го вземе от пакета, с който е получил заявката. На практика този подход е грешен, защото както вече споменахме, не можем да използваме един и същ обект от класа DatagramSocket хем за изпращане, хем за получаване на пакети. Ако използваме два DatagramSocket обекта – един за изпращане и един за получаване, не можем да ги накараме да използват един и същ порт, защото създаването на DatagramSocket обект изисква свободен порт. Ако се опитаме след изпращането на заявката да освободим порта, от който тя е изпратена и веднага да започнем да слушаме на него за отговор, има вероятност през времето, в което още не сме започнали да слушаме, отговорът вече да е пристигнал и да го изпуснем. Както се вижда и този подход не работи. Ето защо силно се

препоръчва въпросът и отговорът при UDP комуникация да са на различни портове. Ето и примерна реализация на клиент за нашия сървър:

```
import java.net.*;

public class UDPDateClient {
    public static void main(String[] args)
    throws Exception {
        // Open an UDP socket for the response
        DatagramSocket socketIn =
            new DatagramSocket();

        // Send request to the UDP Date Server
        DatagramSocket socketOut =
            new DatagramSocket();
        String request = "PORT=" +
            socketIn.getLocalPort();
        byte[] requestBuf = request.getBytes();
        DatagramPacket packetOut =
            new DatagramPacket(requestBuf,
            requestBuf.length,
            InetAddress.getByName("localhost"),
            12345);
        socketOut.send(packetOut);
        socketOut.close();

        // Receive the server response
        byte[] responseBuf = new byte[256];
        DatagramPacket packetIn =
            new DatagramPacket(responseBuf,
                responseBuf.length);
        socketIn.setSoTimeout(5000);
        socketIn.receive(packetIn);
        int len = packetIn.getLength();
        String response =
            new String(responseBuf, 0, len);
        System.out.println(response);
        socketIn.close();
    }
}
```

Както се вижда от кода, клиентът отваря UDP сокет за получаване на отговор от сървъра, подготвя един пакет, като записва в него този порт, след което отваря втори сокет и изпраща през него подготвения пакет на адрес `localhost:12345`, където се очаква да е стартиран сървъра. След това чака за отговор 5 секунди и ако за това време получи пакет с отговор, го отпечатва на конзолата, а в противен случай настъпва изключение.

Multicast сокети

Понякога се налага един пакет да бъде изпратен до много получатели. Обикновено тези получатели са приложения, които са заявили, че искат да получават тези пакети, т.е. те са се абонирали предварително за тях. Да вземем за пример една организация. Служителите в нея са разделени на работни групи и всеки служител иска да комуникира постоянно с колегите от своята група като им изпраща съобщения. От началството пък искат да могат да разпращат важни съобщения до всички служители. Една проста комуникационна система може да се изгради по следния начин: Софтуерът на всеки служител отваря TCP сокет до централен сървър, през който минават всички съобщения. Сървърът знае за всеки служител към кои групи принадлежи и когато получи съобщение предназначено за някоя група, го разпраща до всички нейни членове. По подобен начин, отново с централен сървър, цялата система може да се изгради и чрез UDP сокети. При големи натоварвания, обаче, централният сървър би могъл да не издържи или да работи, но неприемливо бавно. Представете си ако системата служи за комуникация не между хора, а между различни софтуерни компоненти на една сложна информационна система, където има десетки сървъри, като всеки от тях участва в хиляди групи и изпраща стотици съобщения в секунда. Очевидно натоварването е голямо, а рискът от срив на сървъра също не е малък. За решаването на такива проблеми са разработени multicast сокетите. Те много приличат на UDP сокети, но при тях

има три основни операции – абониране за група, изпращане на съобщение до група, отказване от група. Всяка група се идентифицира с уникален IP адрес в локалната мрежа. Един компютър може да е едновременно в много групи. Изпращането на пакет до всички членове на група става като се изпрати този пакет до IP адреса на групата. Работата с multicast групи много прилича на работата с UDP сокети. Да разгледаме един пример – клиентско приложение, което се абонира за групата 224.0.0.1 и отпечатва на екрана всички съобщения, получени в тази група на порт 2002.

```
import java.net.*;

public class MulticastListener {
    public static void main(String[] args)
    throws Exception {
        InetAddress multicastAddr =
            InetAddress.getByName("224.0.0.1");
        byte[] buf = new byte[1024];
        DatagramPacket packet = new
            DatagramPacket(buf,buf.length);
        MulticastSocket multicastSocket =
            new MulticastSocket(2002);
        multicastSocket.joinGroup(
            multicastAddr);
        while (true) {
            multicastSocket.receive(packet);
            String msg = new String(
                packet.getData(),
                0, packet.getLength());
            System.out.println(msg);
        }
    }
}
```

Както виждаме от кода, работата с multicast сокети съвсем не е сложна. Създаваме се multicast сокет, извикваме метода `joinGroup()`, с който се абонираме за някой multicast адрес и след това получаваме в цикъл UDP пакетите, предназначени за тази група и избрания порт. Класът, който използвахме `java.net.MulticastSocket`, има няколко основни метода – `joinGroup()` за присъединяване към multicast група, `leaveGroup()` за напускане на multicast група, `getTTL()` и `setTTL()` за извличане и промяна на параметъра TTL (time to live). Преди да изясним този параметър, трябва да си изясним механизма, по който работи multicasting-а. Груповото разпращане на съобщения (multicasting) се базира на протокола IGMP (Internet Group Management Protocol). Това е мрежов протокол, част от комплекта протоколи TCP/IP, за които говорихме в първата част на настоящия учебен материал. Multicast съобщенията се разпространяват по локална мрежа във вид на IGMP пакети. При подходящи настройки на мрежата тези пакети могат да преминават през маршрутизаторите в локалната мрежа. Преминаването през един маршрутизатор намалява стойността TTL с единица. Достигането на стойност 0 прекратява разпространението на пакета. Стойността TTL означава максималния брой маршрутизатори, през които съобщението може да премине. По стандарт в организацията на IP адресното пространство е предвидена специална зона от адреси от 224.0.0.1 до 239.255.255.255, които са предназначени за multicasting. Всеки от тези адреси би могъл да бъде използван за адрес на multicast група. Абонаментът за multicast услуги става по IP адрес на групата, но в рамките на тази група може да има 65535 различни услуги, съответстващи на различните възможни номера на портове. Поради факта, че за разпространението IGMP пакетите се грижи мрежовия хардуер и софтуер на IP ниво, този метод за разпращане на съобщение до група потребители е изключително ефективен и многократно по-бърз от подходите, които разгледахме по-горе (с централен сървър и TCP или UDP базирана комуникация). Изпращането на един пакет до хиляди компютри абонирани за някой multicast адрес отнема точно толкова време, колкото изпращането на един пакет до един компютър. Това означава, че изпращането на едно UDP съобщение до група получатели по multicast сокет може да е хиляди пъти по-бързо отколкото изпращането на същото съобщение до същата група получатели чрез TCP или UDP. Заради това multicasting-ът е предпочитан метод за работа в локална мрежа на приложения, при които високата производителност е от жизнена важност. Важно е да знаем, че multicasting-ът е свързан с някои особености на локалните мрежи, поради което не е приложим за Интернет. Едно от най-типичните приложения на този вид комуникация е при поддръжката на клъстери от сървъри. При тях multicast-ингът много често се използва за синхронизация на

данните между сървърите, които работят в клъстер, заради високата скорост на разпространение и краткото време необходимо за изпращане на един пакет. Както вероятно повечето от вас знаят, клъстерът представлява група от компютри, които работят заедно като едно цяло. Клъстерът ни дава две основни предимства – скалируемост (scalability) и устойчивост на повреди (fault tolerance). Скалируемостта се постига от това, че имаме възможност да използваме няколко компютъра, които заедно работят много по-бързо отколкото един самостоятелен компютър и като си разпределят по равно работата, повишават многократно производителността на системата. Устойчивостта на повреди осигурява стабилност на системата, така че ако един от компютрите в клъстера се повреди или връзката се него се прекъсне, останалите да поемат неговата работа и клиентите на системата да не разберат за повредата. За да се постигне това, е необходимо в рамките на един клъстер компютрите да са взаимно заменяеми, т.е. всеки от тях да разполага с всичките данни, с които разполагат и останалите, за да може при евентуална повреда на някой от тях да няма загуба на данни. За синхронизацията на тези данни, когато се работи в локална мрежа, най-често се използват multicast сокети, заради огромната им скорост. Освен това при тях отпада необходимостта от централен сървър. Те дават и възможност включването и изключването на компютри в клъстер да може да става по всяко време.
След като изяснихме в общи линии какво е multicasting и за какво се използва, нека да напишем един сървър за клиента, с който започнахме.

```
import java.net.*;

public class MulticastSender {
    public static void main(String[] args)
    throws Exception {
        InetAddress multicastAddr =
            InetAddress.getByName("224.0.0.1");
        MulticastSocket multiSock =
            new MulticastSocket();
        multiSock.joinGroup(multicastAddr);
        while (true) {
            String message = "Hello " +
                new java.util.Date();
            DatagramPacket packet =
                new DatagramPacket(
                    message.getBytes(),
                    message.length(),
                    multicastAddr, 2002);
            multiSock.send(packet);
            Thread.sleep(1000);
        }
    }
}
```

Както виждаме, кодът силно прилича на код, който разпраща UDP пакети. Единствената разлика е, че вместо класа DatagramSocket ползваме класа MulticastSocket и преди да започнем разпращането указваме групата, към която ще пращаме. Дадената сървърска програма в безкраен цикъл праща съобщение, съдържащо текущата дата, с цел да демонстрира получаването на multicast пакетите. За да тестваме клиента и сървъра, ни е необходим компютър включен в локална мрежа. На компютър без мрежова карта и двете програми не работят. Най-добре е ако имаме няколко компютъра и пуснем на всеки от тях няколко клиента и един сървър. Така всеки клиент ще получава всички изпратени пакети, а всеки сървър ще праща до всички клиенти. При желание могат да се добавят и още multicast групи.

Работа с URL ресурси

Още от времената, когато е била предназначена само за писане на аплети, Java се слави с това че, е силно Интернет ориентирана. Извличането на ресурси от Интернет с Java може да бъде изключително лесно, ако използваме класа `java.net.URL`. Нека първо изясним какво URL. Както при пощенските услуги, за доставяне на някаква поща е необходим адрес (държава, град, улица, номер и т.н.), така и в Интернет, в рамките на глобалната информационна система World Wide Web (WWW), за достъп до някакъв ресурс е необходим адрес. Адресите във WWW се наричат

URL. URL е съкращение от Uniform Resource Locator – единен адрес на ресурс и има следния формат:

```
protocol://host[:port]/[resource]
```

Protocol е протоколът, по който е достъпен ресурсът, примерно `http`, `ftp` и др. Host е IP адресът или името на машината, от която е достъпен ресурсът, примерно `www.nakov.com` или `208.185.127.162`. Port е назадължително поле, което указва номера на порта на машината, зададена в полето host, примерно `80` или `8080`. Resource е пълното име на искания ресурс, като се включва и пътя до него. Ако не е зададен, се използва подразбиращият се ресурс. Например ако ресурсът, който искаме да извлечем е http://www.nakov.com/english/CV.html, протоколът е `http`, host-ът е `www.nakov.com`, портът не е зададен и се позразбира че е стандартния за http – `80`, а ресурсът е `english/CV.html`, като `english/` е пътят до ресурса, а `CV.html` е името му.

Ние всекидневно използваме URL адреси за достъп до различни сайтове в Интернет. Когато използваме стандартен Web-браузър, ние пишем в полето за адрес URL-то на сайта, който искаме да посетим и натискаме бутона за извличане на зададения адрес. Когато работим с Java отваряме URL като създаваме обект от класа `java.net.URL`, задаваме адреса на ресурса, от който се интересуваме и го извличаме използвайки стандартен входен поток. Нека илюстрираме това с един малък пример – програма, която извлича документа http://www.nakov.com/english/CV.html и го отпечатва на стандартния изход:

```
import java.net.*;
import java.io.*;

public class URLReaderExample {
    public static void main(String[] args)
    throws Exception {
        URL cv = new URL(
        "http://www.nakov.com/english/CV.html");
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                cv.openStream()));
        String line;
        while ((line = in.readLine()) != null)
            System.out.println(line);
        in.close();
    }
}
```

Кодът е съвсем кратък и ясен – създава се обект от класа URL, като му се подава адреса на ресурса, до който искаме да установим достъп, след това се отваря входен поток за четене на този ресурс с метода `openStream()` и от този поток се прочита целия ресурс ред по ред. Разбира се, разчитаме, че ресурсът е текстов документ и затова го четем с текстов поток. Ако трябваше да извлечем ресурс, който не е текстов, примерно картинка, трябваше да го четем с бинарен поток.

Класът `URL`, заедно с класа `URLConnection` могат да се използват не само за четене на ресурси, но и за писане в ресурси. При писане в ресурс на сървъра, който предоставя този ресурс се изпраща записаната от нас информация, оформена съгласно протокола, по който е достъпен този ресурс. Ето един пример как се прави това:

```
import java.io.*;
import java.net.*;

public class ReverseExample {
    public static void main(String[] args)
    throws Exception {
        String s = URLEncoder.encode(
            "Svetlin Nakov");
        URL url = new URL(
    "http://java.sun.com/cgi-bin/backwards");

        // Send request
        URLConnection connection =
            url.openConnection();
        connection.setDoOutput(true);
        PrintWriter out = new PrintWriter(
            connection.getOutputStream());
        out.println("string=" + s);
```

```
            out.close();

            // Retrieve response
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    connection.getInputStream()));
            String line;
            while ((line = in.readLine()) != null)
                System.out.println(line);
            in.close();
        }
    }
```

В примера се осъществява достъп до услугата http://java.sun.com/cgi-bin/backwards, на която се подава един символен низ, а тя го връща в обратен ред. Тази услуга е реализирана като стандартен CGI script, който се извиква с параметри. Понеже услугата работи по протокола http, е необходимо всичко, което й се изпраща да се кодира съгласно http протокола, така че да бъде разпознато от нея. За целта се използва класа URLEncoder, с който параметрите към скрипта се кодират по стандарта за URL. След това се отваря изходен поток към услугата, подават й се кодираните параметри, после се отваря входен поток и се прочита резултатът от него. За писане в URL се използва класа URLConnection, обект от който се получава с метода `openConnection()`.

Повече информация относно класовете за работа със UDP сокети, multicast сокети и URL можете да намерите в документацията за Java 2 Standard Edition 1.4 на сайта на Sun – http://java.sun.com/j2se/1.4/docs/api/index.html, а също и на сайта на курса "Интернет програмиране с Java" – http://inetjava.sourceforge.net.

JAVA аплети

До преди десетина години беше силно разпространено схващането, че езикът Java служи единствено за създаване на аплети. По това време това схващане беше в голяма степен правилно, защото Java първоначално беше планиран и точно за това и използването му за други цели започна по-късно. Развитието на езика през последните 5-6 години, обаче, коренно промени ролята му в света на програмирането и Java стана една от най-популярните платформи за разработка на корпоративни софтуерни системи. Аплетите, въпреки че вече не заемат централна част в Java платформата, все още си остават една интересна насока в Интернет програмирането, която си струва да разгледаме.

Преди да пристъпим към повече детайли, трябва да си изясним какво е Java аплет. Аплетът е компилирана програма на Java, която се вгражда като обект в обикновена Web-страница и се изпълнява от Web-браузъра по време на разглеждането на тази страница. Аплетите се вграждат в Web-страниците по начин много подобен на вграждането на картинки, но за разлика от тях, те не са просто графични изображения, а програми, които използват правоъгълната област, която броузърът им е дал, за графичния си потребителски интерфейс. Аплетите притежават почти цялата мощ която ни дава Java платформата, но с известни ограничения, предизвикани главно от съображения за сигурност. За да се осигури безопасността на потребителя, на аплетите е позволено да извършват само операции, които не могат да осъществят достъп до потребителската информацията на машината, на която се изпълняват. Аплетите представляват компилирана Java програма във вид на .class файл или съвкупност от компилирани Java класове, записани в .jar файл. Както знаем всички Java програми се изпълняват от Java виртуална машина (JVM) и затова браузърите, които поддържат аплети имат вградена в себе си или допълнително инсталирана виртуална машина. При отварянето на HTML документ с аплети, браузърът зарежда Java виртуалната си машина и стартира аплетите в нея.

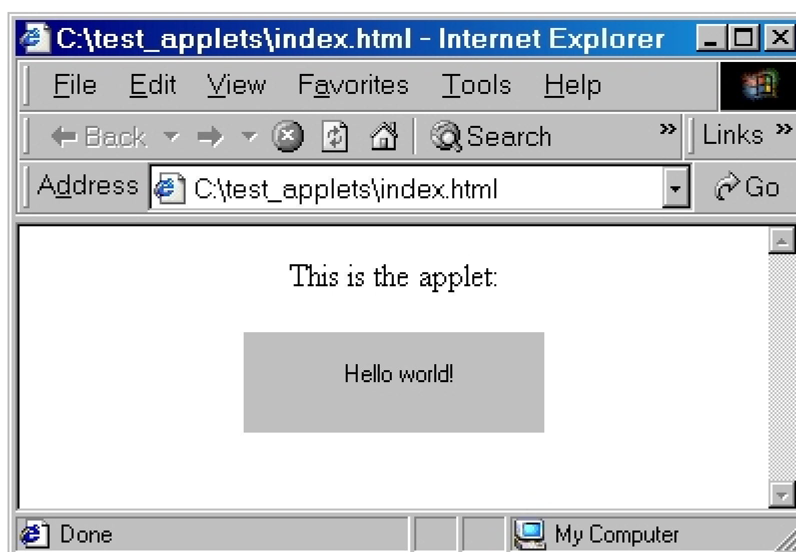
В повечето случаи Java аплетите наследяват класа `java.applet.Applet` и припокриват методите му за инициализация и за изобразяване на екрана – съответно `init()` и `paint()`. В метода `paint()` аплетът изобразява графично на екрана текущото си състояние използвайки стандартните средства на Java за създаване на графичен потребителски интерфейс – AWT (Abstract Window Toolkit). Тези средства се намират в пакета `java.awt` и ще бъдат разгледани по-късно. Да разгледаме един съвсем прост аплет:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorldApplet extends Applet {
    public void paint(Graphics g) {
       g.drawString("Hello world!", 50, 25);
    }
}
```

В този пример единственото, което прави аплетът, е в метода си за изобразяване на екрана да чертае текст в областта, която му е дадена от браузъра на позиция (50, 25) с шрифта по подразбиране. Създаването на аплета `HelloWorldApplet.java` и компилирането му до `.class` файл не е достатъчно за да може той да се изпълни. За разлика от нормалните Java програми аплетите не е задължително да имат `main()` метод. За да видим резултата от нашия аплет трябва да направим Web-страница, в която да го вмъкнем като обект. Ето един пример как става това:

```
<html><body>
    <p align="center">This is the applet:<br><br>
    <applet code="HelloWorldApplet.class"
        width="150" height="50">
    </applet></p>
</body></html>
```

Използвахме HTML тага `<applet>`, в който зададохме името на класа, който искаме да вмъкнем като обект, както и размерите на областта от Web-страницата, която му се предоставя. Ако запишем този HTML код във файла `index.html` и отворим този файл с Internet Explorer, ще получим резултат подобен на следния:



Друг начин да изпълним аплета ни дава програмата `appletviewer`, която е включена към стандартната инсталация на JDK. С нея също можем да изпълняваме аплети, но за разлика от стандартния браузър, appletviewer дава повече права на аплета и работи с текущата версия на JDK. Като параметър appletviewer приема име на HTML файл, който съдържа аплет. Въпросът с версиите на JDK и съвместимостта оставяме за по-нататък, а сега нека навлезем в повече детайли относно това как се създават аплети.

Класът `java.applet.Applet` е базов клас за всички аплети и е наследник на `java.awt.Panel`. Следователно той представлява AWT контейнер и може да се използва за поставяне на различни AWT компоненти с цел изграждане на потребителски интерфейс. Освен за това, той може да се използва също и за директно рисуване със средствата на AWT, както това е направено в горния пример. Класът `java.applet.Applet` дава базовата функционалност на аплетите и предоставя на наследниците си методи за взаимодействие с браузера и външния свят. Да проследим жизнения цикъл на един аплет – какво се случва от момента на зареждане на HTML-страницата с аплета до момента, в който се затвори браузерът или се премине на друга страница. Първоначално браузерът чете HTML документа и намира `<applet>` таговете. За всеки от тях намира и зарежда клас файла, който е указан, инстанцира го в Java виртуалната си машина и започва да го изпълнява. Изпълнението на аплет се състои в следното: Извиква се `init()` метода. Извиква се `start()` метода.

Докато аплетът работи, браузерът му подава всички събития, предназначени за него и го оповестява, когато е необходимо да се пречертае поради някаква причина. Събитията, отнасящи се до аплета, като кликване с мишката, натискане на клавиш и др. се подават на метода `handleEvent()`, който извършва необходимото те да се обработят от AWT контролата, за която са предназначени. Събитията за пречертаване възникват когато се промени видимата част от аплета, например при скролиране на документа или при засичане на аплета с друг прозорец. Обработчикът на тези събития предизвиква извикване на метода `paint()`, при изпълнението на който аплетът е длъжен да се пречертае. При приключване на изпълнението на аплета, браузерът извиква последователно методите `stop()` и `destroy()`. Методът `init()` се извиква еднократно след като аплетът е инстанциран, т.е. е създаден като обект във виртуалната машина на браузъра. В него аплетът може да създаде контролите от потребителския си интерфейс, да направи някои инициализации и всичко останало, което трябва да се изпълни еднократно, преди аплетът да е стартиран. Методът `start()` се вика след инициализацията и след рестартиране на аплета. За разлика от `init()`, методът `start()` може да се извика повече от веднъж. Методът `stop()` се извиква, когато браузерът напуска страницата, в която е зареден аплета. След `stop()` всички нишки на аплета минават в състояние на пауза. При връщане обратно в страницата с аплета, се вика `start()`. Трябва да отбележим че повечето браузъри реализират по различен `start()` и `stop()` методите и затова използването им трябва да става внимателно, а при възможност да се избягва. Методът `destroy()` се извиква еднократно преди аплетът да се унищожи от браузера.

AWT представлява платформено-независима библиотека от класове, която позволява създаване на графичен потребителски интерфейс с Java и дава цялостен компонентно-ориентиран framework за създаване на приложения, които използват графика и взаимодействат активно с потребителя. Този framework ни предоставя стандартен начин за работа с графични компоненти, като бутони, текстови полета, картинки, текст и т.н., а също и механизъм за обработка на събитията възникнали в резултат от взаимодействието между потребителя и програмата, като например щракване с мишката, вход от клавиатурата и т.н. Библиотеката AWT е толкова голяма, че за подробното й описание е необходимо много повече пространство от това, с което разполагаме. Затова без да претендираме за пълнота ще направим само едно кратко частично въведение в AWT и то главно за да изясним това, което ни е необходимо за да пишем аплети. Подробно описание на библиотеката може да се намери на сайта на курса “Интернет програмиране с Java” http://inetjava.sourceforge.net, а също и в документацията на JDK.

Координатната система на аплет с размери sizeX и sizeY започва от позиция (0,0), която отговаря на горния му ляв ъгъл и завършва в позиция (sizeX-1, sizeY-1), която отговаря на долния му десен ъгъл. Изобразяването на графични обекти става чрез класа `java.awt.Graphics`, обект от който се подава на `paint()` метода на аплета. Всеки `Graphics` обект има сбоя собствена координатна система и всеки AWT графичен компонент има свой собствен `Graphics` обект, чрез който реализира визуализацията си. Класът `Graphics` ни дава методи за чертане на основните графични обекти, като линии, правоъгълници, елипси, запълнени многоъгълници, текст с различни шрифтове и много други. Описание на методите, с които се чертаят тези обекти (`drawLine()`, `drawRect()`, `fillRect()`, `clearRect()`, `drawOval()`, `fillOval()`, `drawArc()`, `fillArc()`, `drawPolygon()`, `fillPolygon()` и т.н.) може да се намери в документацията. Освен директното чертане на геометрични фигури, AWT позволява и изобразяване на картинки, заредени от GIF или JPEG файл. За целта се използва класа `java.awt.Image` и метода на класа `Graphics drawImage()`, който има няколко варианта с различни входни параметри. Най-лесният начин за зареждане на картинка в Image обект се дава от метода `getImage()` на класа `Applet`, който приема URL като параметър. Ето един пример как можем да заредим картинка:

```
URL imageURL = new URL(
    "http://www.nakov.com/images/dot.jpg");
java.awt.Image img = this.getImage(imageURL);
```

За да начертаем върху аплета заредената картинка можем да използване следния код:

```
public void paint(Graphics g) {
    g.drawImage(img, 20, 10, this);
}
```

Ако искаме да начертаем картинката с променени размери, можем да използваме отново метода drawImage(), но с други параметри:

```
g.drawImage(img, 0, 0, img.getWidth(this)/4,
    img.getHeight(this)/4, this);
```

Внимателният читател вероятно е забелязал, че методът `drawImage()` има един параметър, за който в нашите примери даваме стойност `this`. Това съвсем не е случайно и се обяснява с архитектурата на AWT и начина, по който се работи с картинки. Методът `drawImage()` приема като последен параметър обект, който реализира интерфейса `ImageObserver`. Зареждането на картинка в AWT винаги е асинхронно, т.е. извършва се паралелно с работата на програмата. Това е съвсем обосновано, като се има предвид, че зареждането на картинка от Интернет отнема известно време, а програмата може да го използва за други цели, вместо да чака. По идея `drawImage()` не изчаква картинката да бъде заредена и тогава да я начертае, а чертае само тази част от нея, която вече е заредена и веднага връща управлението на извикващия метод. Когато картинката се зареди напълно, се извиква методът `imageUpdate()` на интерфейса `ImageObserver`, който трябва да обработи ситуацията по подходящ начин. Обикновено реализацията на метода `imageUpdate()` пречертава картинката. Класът `java.awt.Component`, който е прародител на класа `java.applet.Applet` реализира интерфейсът `ImageObserver` и в метода си `imageUpdate()` пречертава областта от екрана, която е обхваната от картинката, която се е заредила напълно. Използвайки тази базова функционалност на класа `Applet`, можем винаги, когато зареждаме картинки от аплет, да подаваме за `ImageObserver` самия аплет, т.е. обекта `this`. Да разгледаме един цялостен пример за аплет, който използва картинки и реализира проста анимация. Да си поставим за задача направата на аплет, в който една топка постоянно се движи и се отблъсква в стените на аплета при удар. Едно възможно решение на задачата е следното:

```
import java.awt.*;
import java.applet.*;
import java.net.URL;

public class BallApplet extends Applet
implements Runnable {

   public static final int SPEED = 20;
   Image ballImg;
   int x, y, px, py;
   Thread animateThread = null;
   Image bufImg;
   Graphics bufGr;

   public void init() {
       try {
           String imgName =
               getParameter("imgName");
           ballImg = getImage(
               getCodeBase(), imgName);
       } catch (Exception ex) {
           ex.printStackTrace();
           System.exit(-1);
       }
       x = 1; y = 1; px = 1; py = 1;
       bufImg = createImage(
           getSize().width, getSize().height);
       bufGr = bufImg.getGraphics();
   }

   public void paint(Graphics g) {
       if (animateThread != null) {
           // Paint in the buffer
           bufGr.fillRect(0, 0,
               getSize().width, getSize().height);
           bufGr.drawImage(ballImg, x, y, this);
           // Move the buffer to the screen
           g.drawImage(bufImg, 0, 0, this);
       }
   }

   public void start() {
```

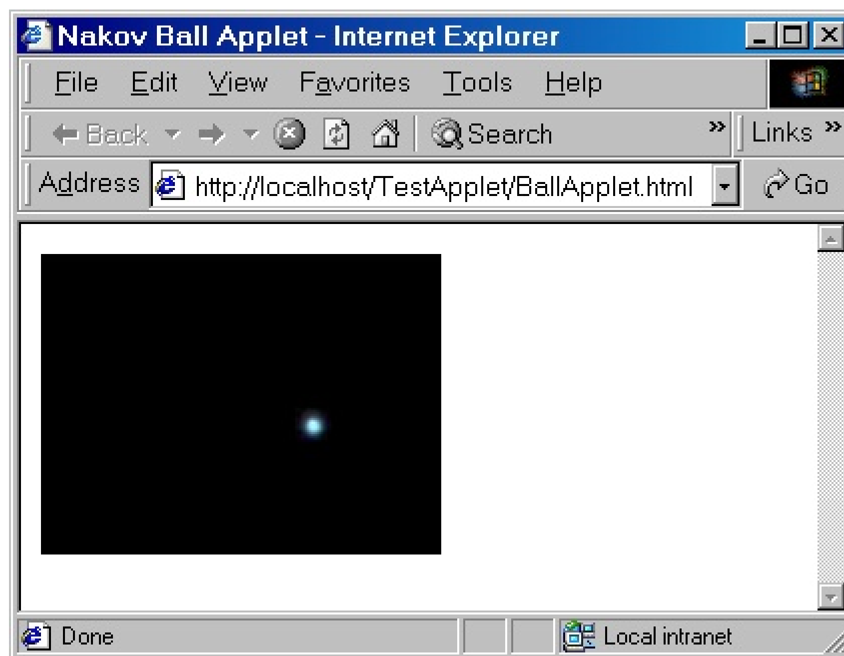
```
        if (animateThread == null) {
            animateThread = new Thread(this);
            animateThread.start();
        }
    }

    public void stop() {
        animateThread = null;
    }

    public void run() {
        // Wait for the image to load completely
        MediaTracker tracker =
            new MediaTracker(this);
        tracker.addImage(ballImg,0);
        try {
            tracker.waitForAll();
        } catch (Exception ex) {}
        // Calculate the animation area size
        int maxX = this.getSize().width -
            ballImg.getWidth(this);
        int maxY = this.getSize().height -
            ballImg.getHeight(this);
        // Animate until interrupt is requested
        while (animateThread != null) {
            if ((x > maxX) || (x < 0))
                px = -px;
            x = x + px;
            if ((y > maxY) || (y < 0))
                py = -py;
            y = y + py;
            try {
                Thread.sleep(SPEED);
            } catch (Exception ex) {}
            // Redraw the applet contents
            paint(getGraphics());
        }
    }
}
```

Идеята за работата на аплета е следната: При инициализация аплетът зарежда картинката с топката и създава един обект `Image`, който ще използва за буфер. При стартиране на аплета се създава една нишка, която се грижи за анимацията. Нейната роля е да променя координатите `x` и `y` на топката съгласно текущата посока на движение, да сменя посоката при удар в стена и след всяка промяна на координатите на топката да пречертава аплета. Пречертаването на аплета работи със специален буфер. Този буфер се използва за да се избегне премигването и да се получи наистина плавно движение. Вси всяко пречертаване на аплета буферът се изчиства с `fillRect()`, след това в него се начертава топката на текущата й позиция и след това на екрана се изобразява съдържанието на този буфер. Тази техника за избягване на премигването при създаване на анимация се нарича двойно буфериране (double buffering). За реализацията й се създава обект от класа `Image` – `bufImg` и се работи чрез неговия `Graphics` обект – `bufGr`. Вместо да се рисува директно върху аплета, се рисува в буфера и готовия кадър от буфера се прехвърля в аплета. При извикване на `stop()` метода аплетът спира нишката за анимация, а при `start()` я създава и я стартира. Името на файла, който съдържа картинката се задава като параметър на аплета и се взема с метода `getParameter()`. Параметрите на аплетите служат за задаване на различни настройки без да е необходима прекомпилация, което се налага ако тези настройки са зададени като константи. За задаването им има специален таг, който се влага в така `<applet>` – тага `<param>`. Ето един примерен HTML код, който стартира нашия аплет и задава за параметъра име на картинка `imgName` стойността `ball.jpg`:

```
<html>
<head><title>Nakov Ball Applet</title></head>
<body>
<applet code="BallApplet.class" width="200"
 height="150">
  <param name="imgName" value="ball.jpg">
</applet>
</body>
```

```
</html>
```

В инициализационната си част аплетът взима параметъра imgName и зарежда картинката с това име от директорията, от която е зареден аплетът. URL, сочещо към тази директория може да се получи чрез метода `getCodeBase()`. Такъв е правилният начин за извличане на ресурс от аплет – не чрез абсолютен URL, а чрез релативен URL спрямо директорията, от която е зареден аплетът. След като е извикан методът за зареждане на картинка, тя е започнала да се извлича от зададения URL, а през това време аплетът създава картинка-буфер за целите на анимацията. При извикване на `start()` метода аплетът създава нишката за анимация и я стартира. При стартиране нишката първо изчаква картинката с топката `ballImg` да се зареди напълно. За целта се използва класа `MediaTracker`, чрез който може да се проследи състоянието на започнали да се зареждат картинки. Да си припомним че работата с картинки в AWT е асинхронна! След като картинката е напълно заредена, можем да сме сигурни, че ако се опитаме да й вземем размерите, операцията ще е успешна. Тези размери, както и размерите на аплета използваме за изчисляване на полето, в което топката може да се движи, така че да не излиза от аплета. Това е правоъгълната област (0, 0) – (maxX, maxY). Самата анимация представлява един цикъл, в който се изчисляват новите координати на топката и аплетът се пречертава с извикване на `paint(getGraphics())`. Между всеки два кадъра от анимацията се изчаква някакво време `SPEED` с цел топката да не се движи прекалено бързо и да се движи с приблизително еднаква скорост на различни компютри. При спиране на аплета с метода `stop()` на променливата `animateThread`, която сочи към нишката за анимация се присвоява `null` и това е знак за `run()` метода на тази нишка, че трябва да приключи работа. Затова в цикъла, в който се движи топката, стойността на `animateThread` се следи непрекъснато. Такъв е препоръчителният начин за прекъсване на работата на нишка. Ето и разултатът от нашия аплет, видян през Internet Explorer:



За да изглежда добре е необходимо картинката, която представлява топката (`ball.jpg`) да е по-малка от размерите на аплета и да бъде на черен фон.

При разработката на аплети и при тестването им с Internet Explorer или друг Web-браузър, трябва да имаме предвид някои неща. Повечето браузъри използват кеширане на различни обекти от Web-страниците, например картинки, стилове и др. за постигане на по-голяма скорост при зареждане на HTML документ. Кеширането се използва и за аплетите, поради което трябва да се внимава. Трябва да се знае, че записването на нова версия на аплета и натискане на бутона “Refresh” не гарантира изпълнението на новата верия. При Internet Explorer сигурен начин за зареждането на последната версия на аплета е натискането на Ctrl+F5 или Ctrl+”Refresh”, но при други браузъри клавишните комбинации са други. Най-сигурно е затваряне на браузъра и стартиране отново. Само така можем да сме сигурни, че последната промяна в кода се е отразила на аплета, и то при условие, че не сме забравили да прекомпилираме.

Друга важна особеност на браузърите е че повечето поддържат много стари версии на JDK. Например Internet Explorer 4.0, 5.0 и 5.5 поддържат само JDK 1.1. Повечето версии на Netscape Navigator също. JDK 1.3.1 се поддържа след издърпване на специален plig-in от сайта на Sun. Поради лошите отношения между Microsoft и Sun, Inetrnet Explorer 6.0 вече не поддържа стандартно аплети, а само след допълнително инсталирване на Java plug-in. Не е ясно защо, но Netscape 6.0 също не подържа стандартно Java аплети и се нуждае от plug-in. Поради изтъкнатите проблеми със съвместимостта когато пишем аплети трябва да използваме JDK 1.1. В противен случай рискът аплетът да не работи на машината на клиента не е малък. Повечето класове и методи, които биха ни потрябвали при писане на аплети ги има в JDK 1.1, така че използването на тази версия не ни ограничава сериозно. Трябва да се съобразяваме, че имената на някои методи в различните версии на JDK са различни, но обикновено има съответствие и съвместимост отдолу нагоре. Например в `java.awt.Component` от JDK 1.2 е въведен метод `getWidth()`, а в JDK 1.1 вземането на широчината на компонент става с `getSize().width`. При използването на липсващ метод на някой клас, виртуалната машина на браузера предизвиква изключение. Изключенията, които възникват в аплетите, както и всичко, отпечатано чрез `System.out.println()` можем да видим в Java конзолата на браузъра. В повечето браузъри тя е достъпна от менюто. В Internet Explorer Java конзолата се появява в менюто "View" само след като се разреши от опциите (Internet Options | Advanced | Microsoft VM | Java console enabled) и се рестартира браузерът. Намирането на проблем в аплет без Java конзолата е немислимо, така че когато разработвате аплети и нещо не работи, винаги поглеждайте в конзолата. Типичен е случаят, в който в средата за разработка (например Jbuilder или IDEA) аплетите работят, а в браузера не искат. Причините са две – разлика във версиите на JDK и ограничените права, които се дават на аплетите. Обикновено `appletviewer` или средата за разработка стартират аплетите с повече права, отколкото един браузър би им дал и с JDK, различно от 1.1 и затова се получава несъвместимост. Преди да изясним ситуацията с правата на аплетите, трябва да отбележим, че в новите версии на JDK съществува разширение на библиотеката AWT, което се нарича Swing и се намира в пакета `javax.swing`. Използването на Swing в аплети не се препоръчва, защото много малко браузъри поддържат Swing стандартно.

В предходната част от курса започнахме темата за създаването на Java аплети. В тази част ще довършим изложението по тази тема и ще продължим с темата "Web-приложения".

JAVA аплети и сигурност

Сигурността на аплетите е важна тяхна черта. Никой потребител не би се съгласил да разглежда сайтове с аплети, ако те могат да пишат свободно по диска му, ако могат да откраднат негова лична информация, да изпращат email-и от негово име или да извършват някаква друга злонамерена дейност. Ето защо по идея аплетите работят с ограничени права. Сигурността в Java е част от самата платформа и се конфигурира от един специален файл с име `java.policy`. В зависимост от правата, които Web-браузърът иска да даде на аплета, се подготвя съответен файл, който ги описва и виртуалната машина се конфигурира по него. В някои браузъри правата могат да се настройват и се допуска възможност потребителят да дава пълно доверие на определени сайтове, с което аплетите се освобождават от ограниченията си. Ако потребителят не е посочил нещо друго, се използват стандартните настройки за правата на аплетите, които налагат следните ограничения: Аплетите не могат да четат и пишат по диска на машината, на която се изпълняват. Не могат да осъществават достъп до чужда памет, дори в операционни системи, в които няма защита на паметта. Не могат да отварят сокет до произволен сървър в Интернет. Могат да отварят сокет само до хост-а, от който са заредени. Не могат да извикват директно native код. Не могат да предизвикат претоварване или забиване на машината, на която се изпълняват. Последното е възможно да се случи в някои специфични ситуации, но това се дължи на грешки и пропуски в сигурността на съответните браузъри и виртуалните машини, които те използват. Трябва да обърнем специално внимание на сокетите. Свидетели сме на много аплети, които извършват активна мрежова дейност, като например аплети за Chat, аплети за четене на email, аплети за изпращане на email, различни игри и т.н. Всички те използват сокет базирана комуникация и

изглежда, че отварят сокет към Интернет. Например при изпращането на поща аплетът комуникира със зададен от потребителя SMTP сървър. Това, обаче, не става директно, както при обикновените програми на Java. Аплетите имат право да се свързват чрез сокет само до сървъра, от който са заредени, т.е. към хост-а върнат от метода `getCodeBase().getHost()`. Ето защо аплети, които не са заредени от някой Web-сървър, а локално от файловата система, чрез отваряне на локален HTML файл, нямат право да отварят никакви сокети. Това защитава потребителите от атака чрез HTML документи съдържащи аплети със злонамерено действие. Всички аплети, които изглежда, че отварят сокети към Интернет, всъщност отварят сокети към сървъра, от който са заредени и от там получават пренасочване към заявения хост, т.е. използват Web-сървъра като прокси (междинен пренасочващ сървър). Когато се наложи да пишем аплет, който комуникира чрез сокети, е необходимо на Web-сървъра, където се хоства този аплет да пуснем някакъв допълнителен сървър, който осигурява комуникацията на аплета с услугата, до която той трябва да осъществява достъп. Разбира се, това трябва да става след успешна автентикация на потребителя в системата. За целта най-удобно е сървърът, който се грижи за комуникацията на аплета да се интегрира в Web-сървъра, за да може да използва информацията от сесията на потребителя, който е изпълнил аплета. Как точно може да стане това ще изясним по-нататък в нашия курс в частта за Web-приложения.

Нека сега дадем още един пример за аплет, с който да демонстрираме работа с компонентите на AWT и обработката на събитията, възникнали в резултат от действията на потребителя. Да си поставим за задача реализацията на прост калкулатор, който събира числа. Трябват ни две полета за двете събираеми, още едно поле за резултата и един бутон за събиране. За демонстрация на работата със шрифтове ще добавим в нашия аплет-калкулатор заглавен текст със сянка. За демонстрация на работата със събития от мишката при щракване върху аплета цветът му ще се променя, а при отпускане бутона на мишката ще се възстановява обратно. Ето една примерна реализация на аплета:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class SumatorApplet extends Applet {
    TextField number1Field = new TextField();
    TextField number2Field = new TextField();
    Button calcButton = new Button();
    TextField sumField = new TextField();
    Color lastBackground;

    public void init() {
        this.setBackground(Color.black);
        // Set layout manager to null
        this.setLayout(null);
        // Create the first text field
        number1Field.setBounds(
            new Rectangle(20, 50, 60, 25));
        number1Field.setBackground(Color.white);
        this.add(number1Field, null);
        // Create the second text field
        number2Field.setBounds(
            new Rectangle(95, 50, 60, 25));
        number2Field.setBackground(Color.white);
        this.add(number2Field, null);
        // Create the "calculate sum" button
        calcButton.setBounds(
            new Rectangle(170, 50, 90, 25));
        calcButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(
                    ActionEvent e) {
                        calcSum();
                }
            });
        calcButton.setLabel("calc sum");
        this.add(calcButton, null);
        // Create the result text field
        sumField.setEditable(false);
```

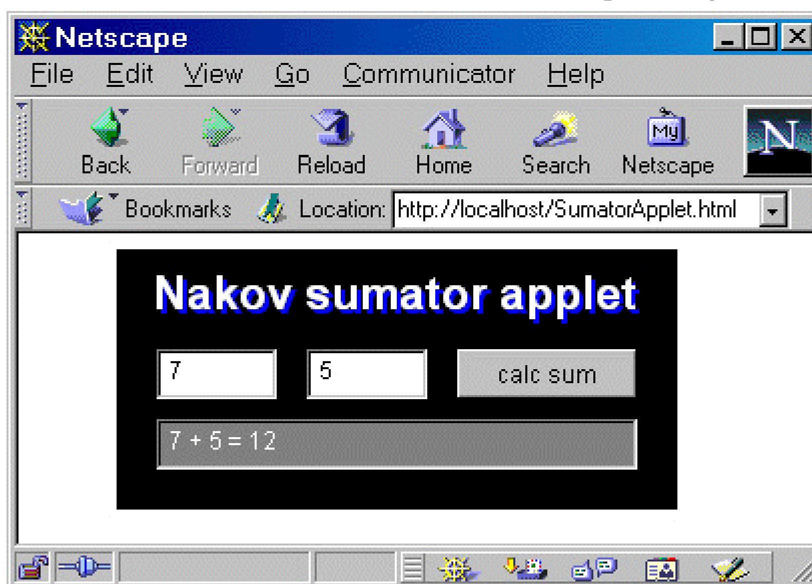
```
            sumField.setBackground(Color.gray);
            sumField.setForeground(Color.white);
            sumField.setBounds(
                new Rectangle(20, 85, 240, 25));
            this.add(sumField, null);
        }

        public boolean mouseDown(Event evt,
        int x, int y) {
            lastBackground = this.getBackground();
            this.setBackground(Color.red);
            return true;
        }

        public boolean mouseUp(Event evt,
        int x, int y) {
            this.setBackground(lastBackground);
            return true;
        }

        private void calcSum() {
            try {
                long s1 = new Long(number1Field.
                    getText()).longValue();
                long s2 = new Long(number2Field.
                    getText()).longValue();
                sumField.setText(s1 + " + " +
                    s2 + " = " + (s1+s2));
            } catch (Exception ex) {
                sumField.setText("Error!");
            }
        }

        public void paint(Graphics g) {
            super.paint(g);
            Font font = new Font(
                "Dialog", Font.BOLD, 23);
            g.setFont(font);
            g.setColor(Color.blue);
            g.drawString(
                "Nakov sumator applet", 20, 32);
            g.setColor(Color.white);
            g.drawString(
                "Nakov sumator applet", 18, 30);
        }

        public static void main(String[] args) {
            Frame frame = new Frame("Sumator");
            frame.setSize(280,160);
            SumatorApplet applet =
                new SumatorApplet();
            applet.init();
            frame.add(applet);
            frame.setVisible(true);
            applet.start();
        }
    }
```

В инициализационната част на аплета първо се задава стойност `null` за Layout Manager. Layout Manager-ът служи за подреждане на AWT компонентите в един AWT контейнер и може да е много полезен при създаване на форми, които могат да променят размерите си. В случая искаме да работим с абсолютни координати и размери на компонентите, а не с размери и координати, определени от Layout Manager-а и затова задаваме за LayoutManager стойност null, понеже по подразбиране тази стойност е друга. След това създаваме компонентите една по една – първото текстово поле, второто текстово поле, бутонът. За да прихванем събитието "натискене на бутона за сумиране", използваме методът `addActionListener`, който приема като параметър обект от клас, който имплементира интерфейса `ActionListener`. За спестяване на някои неудобства използваме дефиниция на място на анонимен клас, който реализира ActionListener интерфейса и в метода за натискане на бутон `actionPerformed()` извиква метода за изчисляване на сумата `calcSum()`. За прихващане на събитията от мишката има два начина. Единият, който ние сме използвали е да се

припокрият методите `MouseDown()`, `MouseUp()` и т.н. на базовия клас, а другият е да се добави MouseListener с метода `addMouseListener()` по начин подобен на този с добавянето на ActionListener. Процедурата за пресмятане на резултата взима стойностите от двете текстови полета, превръща ги в числа и показва резултата от събирането в полето за резултата. Ако не успее при превръщането тескта от полетата в числа или ако се случи препълване или някаква друга грешка, резултатът е "Error!". За демонстрация на работата с шрифтове в метода paint() се отпечатва текст със сянка. Важно е методът `paint()` да извиква `paint()` метода на базовия си клас (`super.paint()`), за да могат компонентите, добавени в аплета да се пречертават всеки път, когато аплетът се пречертава. В нашата реализация на `paint()` метода след извикването на базовия `paint()` метод, чертането продължава със задаване на шрифта и цвета на текста и отпечатването му. След това цветът се сменя и се отпечатва същия текст, изместен с 2 позиции нагоре и наляво. Така се създава впечатлението за сянка. Нашият аплет има още една интересна възможност – може да работи и като самостоятелна програма. За целта той има `main()` метод, в който се създават аплета и един обект `java.awt.Frame`, след което във Frame-а се поставя аплета, задават му се размерите и се показва на екрана. Така аплетът може да бъде стартиран за тестови цели като самостоятелна програма, а когато е готов, може да се тества и в браузера, защото, както знаете аплетите се държат по различен начин в различните среди, в които се стартират. Ето примерен HTML код, с който се стартира аплета:

```
<html><body><center>
<applet code="SumatorApplet.class"
    codebase="." width="280" height="130">
</applet>
</center></body></html>
```

Забелязваме параметъра codebase=".". С него може да се задава пътя до директорията, в която се намира .class файла на аплета. Тагът <applet> има и други параметри, например `ARCHIVE="SomeArchive.jar"`, с който може да се зададе пътя и името на JAR архив, който съдържа класовете на аплета. Ето как изглежда нашият аплет под Netscape Navigator:



В Интернет е пълно със сайтове, които активно използват аплети. Типичен пример за такъв сайт е Web версията на ICQ, достъпна от адрес http://icqlite.icq.com. Липсата на поддръжка за по-високи версии на JDK от 1.1 ни навежда на мисълта, че аплетите са малко остаряла и изоставена технология. Това е вярно в някаква степен и то главно заради отказа на Microsoft да поддържа Java. От няколко години насам Internet Explorer е най-масово използваният браузър и той поддържа само JDK 1.1. До момента, обаче, няма масово навлязла технология която да замести аплетите и затова те си остават единственото решение на много проблеми при разработката на Web-базиран софтуер. Единствената масово навлязла технология в тази насока е Macromedia Flash, която е предназначена главно за мултимедия. Тя, обаче, няма същата мощ, която има Java и поради това не може да замести аплетите.

Повече информация за AWT, разработка на аплети, примери и други материали можете да намерите на сайта на курса: http://inetjava.sourceforge.net.

По идея нашият курс по "Интернет програмиране с Java" има за цел изясняването на три основни технологии: сокет програмиране, аплети и Web-приложения. До момента разгледахме първите две от тях – многонишковото програмиране с TCP, UDP и Multicast сокети и разработката на Java аплети. Предстои ни последната и най-важна част от курса – разработката на Web-приложения.

Web-приложения с JAVA

Web-приложенията представляват информациони системи, които са достъпни през Интернет или локална мрежа чрез стандартен Web-браузър. Например всички Web-базирани системи за електронна поща (като `mail.yahoo.com`, `abv.bg`, `mail.bg` и др.) представляват Web-приложения. За достъп до една Web-базирана система, е необходимо потребителят да разполага със стандартен Web-браузър (например Internet Explorer) и връзка с компютъра, на който се намира тази система. Обикновено връзката се осъществява чрез Интернет, а за достъп до системата се използва адресът на нейния Web-сайт в рамките на глобалната информационна система WWW. Глобалната разпределена информационна система WWW (World Wide Web) представлява съвкупността от всички сървъри в Интернет, предоставящи достъп до ресурси чрез стандартен Web-браузър. Огромно е разнообразието от технологии, на които тя е изградена. Огромни са и възможностите за взаимодействие между сървърите, които я изграждат. За публикуването на информация в Web пространството (WWW) се използват Web-сървъри. Web-сървърът представлява софтуер, който предоставя достъп до някаква информация чрез протокола HTTP. Web-сървърите могат да предоставят както статични ресурси, така и ресурси, които се създават в момента на заявката за достъп до тях (динамично генерирана информация). На един Web-сървър може да има един или няколко Web-сайта, всеки от които е комбинация между статично и динамично съдържание. Един сайт, разбира се, може да е разположен на няколко Web-сървъра. Едно Web-приложение представлява софтуерна система с Web-базиран потребителски интерфейс, работеща на някакъв Web-сървър в Интернет или в локална мрежа. На един Web-сървър може да има няколко независими Web-приложения, както и статична информация, която не е обвързана с никое от тях. Web-приложенията могат да взаимодействат по между си по най-разнообразни начини. Възможно е резултите от няколко независими Web-приложения, работещи на различни и отдалечени един от друг сървъри да се използват съвместно в една Web-страница. Типичен пример за такова взаимодействие са рекламите по сайтовете в Интернет. Например приложението за електронна поща достъпно от `mail.yahoo.com` показва в един HTML документ резултата от работата на две различни Web-приложения. Едното е приложението за четене на поща, което дава потребителски интерфейс за четене и изпращане на писма, а другото е приложението, което се грижи за рекламите и се изпълнява на съвсем друг сървър. Друг пример за съвместно използване на няколко Web-приложения са броячите на посетитети, които се вграждат в различни сайтове и представляват независими приложения, обикновено работещи на отделни сървъри.

Обикновено един Web-сайт представлява съвкупност от статично съдържание (Web-страници, картинки, документи и др.), динамичнно съдържание (Web-страници и други документи) и Web-приложения. Както статичното, така и динамичното съдържание в един сайт може да е разпределено на различни сървъри. Трудно е да се дефинира точната граница между две Web-приложения, защото и те могат да бъдат разпределени на няколко сървъра и да работят като една цяла система.

Най-често под Web-приложение се разбира цялостна софтуерна система за предоставяне на някаква услуга на потребителя през Web. От гледна точка на програмирането Web-приложенията представляват стандартни клиент-сървър системи. Клиентът, както вече знаем, е стандартният Web-браузър, а сървърът е Web-сървърът, на който работи Web-приложението. Характерна черта за Web-приложенията е, че към тях осъществяват достъп много потребители едновременно. Всеки

потребител се обслужва назависимо от другите потребители, така сякаш е единствен. Друга характерна черта на Web-приложенията е, че работят с еднопосочна комуникация, на принципа заявка-отговор (request-response). Браузърът на клиента дава заявка за някакъв ресурс и сървърът отговаря на тази заявка с изпращането на поискания ресурс или със съобщение за грешка. Този модел на комуникация лишава сървъра от възможността да изпраща асинхронно данни на клиента по свое желание. Това ограничение сериозно затруднява системите, които осъществават достъп до информация в реално време.

Web-приложения могат да се разработват на различни езици и за различни Web-платформи – CGI, Perl, PHP, ASP, Java/JSP и др. При разработката на Web-приложения с Java се използват технологиите Java-сървлети и Java Server Pages (JSP) и платформата за Web-приложения на Sun, която е част от J2EE (Java 2 Enterprise Edition). Тази платформа ни дава стандартен framework (съвкупност от програмни средства и стандарти) за разработка на Web-приложения, който ще разгледаме по-късно. Нека започнем с основите концепции на Web-програмирането.

Web-сървър

От гледна точка на Интернет програмирането Web-сървърите са приложения, които "слушат" на определен порт (обикновено това е стандартният порт за HTTP – 80), и отговарят на HTTP заявките, получени от клиентски приложения (най-често това са Web браузърите). Простите Web-сървъри могат само да връщат в отговор на заявките файловете, които са разположени в директорията, обозначена като главна Web-директория. Например ако имаме един прост Web-сървър стартиран на компютъра с Интернет адрес `www.mywebserver.com` и сме указали, че главната му директория е `C:\MyWebSite`, то когато Web-браузърът поиска ресурса `http://www.mywebserver.com/pictures/index.html`, нашият прост Web-сървър ще му предостави файла `C:\MyWebSite\pictures\index.html` (ако съществува). Всички съвременни Web-сървъри имат възможността да предоставят на клиентите си не само файлове от главната Web-директория и нейните поддиректории, но и динамично генериран HTML, получен от работата на външна за Web-сървъра програма. Тази технология се нарича CGI (Common Gateway Interface). При CGI на базата на HTTP заявката Web-сървърът стартира някоя CGI-програма и връща на клиента това, което тази CGI-програма изпише на стандартния изход като резултат от изпълнението си. CGI-програмата може да бъде написана на практически всеки език или script за програмиране (например на C, C++, Pascal, Perl, PHP и др.). Има и други възможности за динамично генериране на HTML – например не чрез външна програма, а чрез модул вграден директно към Web-сървъра. Такъв подход използва ISAPI технологията на Microsoft, която дава възможност за динамично вграждане в сървъра на компилиран програмен код от DLL файл. Технологията, която лежи в основата на Web-програмирането с Java – JSP/Servlets, също използва вграждане в Web-сървъра на компилиран програмен код (Java класове), който генерира динамично HTML. Вграждането на компилиран програмен код пред извикването на външна програма има известни предимства. По скорост на изпълнение е по-ефективно, защото не се налага за всеки динамично генериран HTML документ да се извиква външно приложение, при което в операционната система се създава нов процес. Интеграцията между сървъра и вградения в него компилиран код е по-лесна отколкото интеграцията с външна CGI-програма.

HTTP протокол

Не е разумно да говорим за Web-програмиране преди да сме изяснили протокола по който си комуникират Web-сървърите и Web-браузърите – HTTP (Hyper Text Transfer Protocol). HTTP представлява прост текстов протокол, който се използва за пренос на практически вякакъв вид данни, наричани събирателно **ресурси**. В HTTP протокола има понятия като клиент (обикновено това са Web-браузърите) и сървър (това са Web-сървърите). Обикновено HTTP протоколът работи през стандартен TCP сокет отворен от клиента към сървъра. Стандартния порт за HTTP протокола е 80, но може да се използва и всеки друг свободен TCP порт. Комуникацията по HTTP се състои

от заявка (request) – съобщение от клиента към сървъра и отговор (response) – отговор на сървъра на съобщението от клиента.

HTTP заявки

HTTP заявката има следния формат:

```
<метод> <URI> HTTP/1.1
<header-полета>
<празен ред>
```

Забелязват се 3 основни елемента: **метод**, **URI** и **header-полета**. **Методът** описва вида на HTTP заявката, изпратена от клиента. Най-често използваните методи са GET и POST. Чрез GET клиентът изисква някакъв ресурс от Web сървъра. POST служи за предаване на данни към сървъра. Имената на методите в HTTP заявките се изписват винаги с главни букви. Уникалният идентификатор **URI (Unique Resource Identifier)** еднозначно определя ресурса, над който ще оперира заявката. Това е частта от URL, която стои след името на хост-а (сървъра) в URL-то. Фрагментът **HTTP/1.1** с който завършва първият ред задава версията на HTTP протокола, която ще бъде използвана за осъществяването на HTTP сесията. **Header-полетата** от заглавната част задават допълнителни параметри на заявката и определят различни изисквания относно ресурса, който се очаква да бъде върнат от Web-сървъра. **Празният ред** определя края на заявката. Да дадем един пример за HTTP заявка, която връща началната страница от сайта `http://www.dir.bg/`:

```
GET / HTTP/1.1
Host: www.dir.bg
↵
```

Методи на HTTP заявката

Протоколът HTTP версия 1.1 поддържа общо 8 различни метода: GET, POST, HEAD, PUT, DELETE, OPTIONS, TRACE, CONNECT. Най-често използваните методи са GET и POST и те имат най-голямо отношение към Web-програмирането.

GET методът преставлява команда за извличане на ресурс, указан от зададено URI. Всичко, което прави Web-сървърът за извличането на статичен ресурс чрез GET заявка е да го прочете от файловата система и да го върне на клиентите в подходящ HTTP отговор. При извличане на динамичен ресурс сървърът извиква компилираният програмен код, който генерира ресурса и връща резултата от него в HTTP отговор. Ето един реален пример за HTTP заявка с GET метод:

```
GET /InetJava-2002-program.html HTTP/1.1
Host: inetjava.sourceforge.net
Accept: */*
Accept-Language: bg
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0(compatible;MSIE 6.0; Windows NT 5.0)
Connection: Keep-Alive
Cache-Control: no-cache
↵
```

Изпращането на тази заявка към Web-сървъра, който слуша на порт 80 на машината с Интернет адрес `inetjava.sourceforge.net`, ще върне файла `InetJava-2002-program.html`, достъпен от URL `http://intejava.sourceforge.net/InetJava-2002-program.html`. Ако към искания ресурс трябва да се зададат параметри, това става като към URI-то се добави въпросителен знак, а след него двойки от вида `<име на параметър>=<стойност>`, като двойките от този вид се разделят една от друга със `&`. За избягването на някои непозволени символи се използва така нареченото URL-кодиране, за което ще говорим по-късно.

POST методът служи за изпращане на данни от клиента към Web-сървъра. Обикновено сървърът предава получените от POST заявката данни на някакъв CGI скрипт или вграден модул за динамично генериране на HTML, който ги обработва и връща някакви резултати. Тези резултати се връщат на клиента като отговор на неговата заявка. Ето и един реален пример за HTTP заявка с POST метод, изпратена от Internet Explorer 6.0 при опит за влизане в Web-базираната система за електронна поща на `www.abv.bg`:

```
POST /webmail/login.phtml HTTP/1.1
Host: www.abv.bg
Accept: */*
Accept-Language: bg
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0(compatible;MSIE 6.0; Windows NT 5.0)
Connection: Keep-Alive
Cache-Control: no-cache
Content-Length: 59
↵
LOGIN_USER=boris
DOMAIN_NAME=abv.bg
LOGIN_PASS=tajnamajna
↵
```

Както се вижда, параметрите се предават след самата заявка, като в header-полетата се указва общата дължина в символи на всички параметри и техните стойности. За разделител между header-полетата и параметрите се използва празен ред. За край на заявката се използва също празен ред. Ако параметрите съдържат непозволени символи, те се кодират по специален начин. Кодирането и разкодирането на параметрите и стойностите им се прави автоматично от Web-браузъра и Web-сървъра. Като Web-програмисти не е необходимо да знаем в детайли точно как става това.

Отговори на HTTP заявки

На всяка HTTP заявка, независимо дали е валидна или не, Web-сървърът връща някакъв отговор. При валидна заявка за съществуващ ресурс Web-сървърът връща този ресурс, а в противен случай връща грешка. Отговорът на HTTP заявка има следния формат:

```
HTTP/1.1 <код> <текст>
<header-полета>
<празен ред>
<ресурс>
```

Първият ред се нарича **статус линия** и съдържа версията на HTTP протокола, по който се изпраща отговора, трицифрен код на резултата или код на грешка и кратко текстово описание на този код. Следват **header-полетата.** Те съдържат различни параметри на върнатия ресурс, както и информация за Web-сървъра. Следва **празен ред**, а след него **ресурса**, кодиран по описания в header-полетета начин. В зависимост от типа на ресурса, сървърът може да го върне кодиран по различен начин и по различен начин да укаже на клиента колко байта е дълъг HTTP отговорът. Стойностите на header-полетата и формата на ресурса са от интерес основно за Web-браузъра и затова няма да ги разглеждаме в детайли. Основното, което трябва да знаем е, че на всяка HTTP заявка сървърът отговаря с HTTP отговор, който съдържа искания ресурс или грешка. Кодовете на грешките започват с цифрата 4 или 5. Кодовете на успешен резултат започват с 2, а кодовете, носещи специална информация – с 3. Най-често срещаните кодове при HTTP отговор са: 200 – успех; 304 – документът не е променян от времето, зададено в header-а (използва се от браузърите за кеширане на документи); 404 – ресурсът не е намерен; 500 – грешка на сървъра. Ето един пример за изпращане на HTTP заявка за извличане на главната страница от локално стартиран Web-сървър и отговорът на тази заявка:

```
C:\> telnet localhost 80
GET / HTTP/1.1
Host: localhost
↵

HTTP/1.1 200 OK
Date: Sat, 10 Aug 2002 16:09:18 GMT
Server: Apache/1.3.9 (Win32)
Accept-Ranges: bytes
Content-Length: 73
Content-Type: text/html
↵
<html>
<head> <title> Test </title> </head>
    Test HTML page.
</html>
```

Както се вижда, сървърът е върнал отговор на HTTP заявката с код 200 (успех) и върнал искания ресурс. Ето и един пример за неуспешно завършила заявка:

```
C:\> telnet localhost 80
GET /img/nakov.gif HTTP/1.0
↵

HTTP/1.1 404 Not Found
Date: Sat, 10 Aug 2002 16:20:17 GMT
Server: Apache/1.3.9 (Win32)
Connection: close
Content-Type: text/html
↵
<HTML><HEAD>
<TITLE>404 Not Found</TITLE>
</HEAD><BODY>
<H1>Not Found</H1>
The requested URL /img/nakov.gif
was not found on this server.<P>
<HR><ADDRESS>Apache/1.3.9
Server at test Port 80</ADDRESS>
</BODY></HTML>
```

Вероятно сте забелязали, че в последния пример използваме заявка по протокол HTTP/1.0, а в предходния – по HTTP/1.1. Най-съществената разликата между двете версии на протокола е, че при HTTP/1.0 след връщането на отговора на HTTP заявка сървърът веднага затваря сокета с клиента, а при HTTP/1.1 може с едно отваряне на сокет да се изпълнят последователно няколко HTTP заявки. Това прави HTTP/1.1 протоколът по-бърз заради което е предпочитан от повечето HTTP клиенти.

В предходната част от курса започнахме темата за създаването на Web-приложения с Java. В тази част ще продължим с тази тема. Ще изясним основните концепции в Web-програмирането, ще обясним какво е Java сървлет, как се създават и как се изпълняват сървлети със сървъра Tomcat.

Основни концепции в Web-програмирането

Всички сме виждали Web-базирани e-mail системи като mail.yahoo.com, mail.bg и abv.bg. Те са чудесни примери за Web-приложения. Както знаем от предходната част на настоящия задочен курс, Web-приложенията представляват програмни системи, които работят на някакъв Web-сървър и предоставят на потребителите Web-базиран интерфейс. Комуникацията между потребителите и Web-приложенията се основава на заявки и отговори и се извършва по протокол HTTP. Когато потребителят напише адреса на някое Web-приложение, неговият Web-браузър изпраща на съответния Web-сървър заявка за достъп до това Web-приложение и получава динамично генериран отговор във вид на HTML или друг формат, който браузърът разбира. Ще се опитаме да представим различните аспекти на Web-програмирането, неговите предимства и недостатъци. Също така ще направим една уговорка – като средства за изграждане на потребителския интерфейс на Web-приложенията ще имаме предвид само HTML, CSS и JavaScript. Технологии като Flash, ActiveX и Java-аплети няма да бъдат разглеждани в останалата част от този курс. Можем да разделим условно Java-базираните Web-приложения на две-части:

- сървърска част – представлява съвкупност от Java сървлети и JSP-та, които обработват получените от потребителя данни и взависимост от тях динамично генерират HTML документи, CSS стилове и JavaScript код
- клиентска част – представлява съвкупността от динамично генерираните HTML документи, CSS и JavaScript код, които се визуализират от Web-браузъра и изграждат потребителския интерфейс на приложението

Разглеждайки Web-приложенията по този начин, можем да ги определим като многопотребителски клиент-сървър приложения, предназначени за работа в Интернет или Интранет.

HTML

HTML (Hyper Text Markup Language) е създаден като част от WWW (World Wide Web) от Тим Бернерс-Лий в началото на 90-те години. HTML е базиран на SGML (Standard Generalized Markup Language – стандартен формат за представяне на текст, широко използван от американското правителство) и наследява неговия синтаксис. HTML не е програмен, а описателен език за представяне на форматиран текст. HTML документите представляват изцяло текстови файлове, като в тях освен текста, който съдържат, са вмъкнати инструкции за форматиране (наречени тагове), които указват как точно да се изобрази текста, по време на визуализацията. В HTML документите могат да се указват връзки (hyperlinks) към произволни отдалечени ресурси. Въпреки че в последно време HTML претърпя доста бурно развитие, което доведе до неговото усложняване, основната му сила си остава неговата простота. Когато се използва в комбинация с различни технологии като JavaScript и CSS, езикът HTML предлага доста богати възможности за реализиране на потребителския интерфейс на сложни Web-приложения.

CSS

Cascading Style Sheets е допълнение към HTML. Разработен е от W3C (World Wide Web Consortium) и представлява език за описание на атрибутите и позиционирането на елементите на HTML документи. Чрез CSS се дефинират стилове, които се използват след това в HTML документите за форматиране на текста. При необходимост форматирането на един HTML документ, използващ CSS, може бързо и лесно да се промени, като се променят само стиловете в CSS файла, без да се променя HTML файла.

JavaScript

JavaScript е сравнително прост скриптов език, който се изпълнява от Web-браузъра на потребителя и позволява динамична манипулация на обектите в HTML документите. С негова помощ е възможно създаването на сложни по функционалност и интересни Web-страници. Първоначално е разработен от Netscape, но в момента се поддържа в една или друга степен от всички браузъри.

Разпределеност и платформена независимост на Web-приложенията

Най-голямото предимство на Web-програмирането е, че Web-приложенията са достъпни от различни компютри, работещи под различни операционни системи и различни браузъри, като единственото условие е те да имат достъп до Web-сървъра, където работи съответното Web-приложение. Всичко което е необходимо на потребителя е връзка с Интернет и стандартен Web-браузър. Поради независимостта от операционната система на потребителя и относителната независимост от неговия Web-браузър, Web-програмирането улеснява работата на софтуерните разработчици, като им дава възможност да поддържат само една версия на приложението, която работи на всички платформи, а не отделни версии за всяка отделна платформа. Като се има предвид разнообразието от операционни системи в Web-пространството, това че се поддържа само една версия на приложението е едно сериозно предимство пред другите видове програмиране.

Понякога, в стремежа си да създадат по-сложни Web-приложения програмистите се възползват от възможностите на определен браузър. Често срещана практика е да се оптимизира едно приложение за Microsoft Internet Explorer. Той поддържа голяма част от общоприетите спецификации за HTML и CSS, но има и множество специфични, нестандартни тагове и стилове. Ако се използват нестандартните стилове и тагове, има голям риск приложението да не работи с други браузъри. Доста е трудно да се направи нещо сложно с JavaScript, което да работи еднакво добре както на Internet Explorer, така и на Netscape Navigator и Opera. Подобен проблем се появява

понякога и от това, че самият HTML не изглежда еднакво на всички браузъри, макар и да не са използвани нестандартните възможности на някой конкретен браузър.

Допълниетелно улеснение на Java разработчиците е платформената независимост на Java. Благодарение на нея не само клиентската, но и сървърската част на Web-приложенията става напълно платформено независима. Вградените възможности на Java за унифициран достъп до бази данни през JDBC или EJB допълнително улесняват работата на програмиста и го правят по-малко обвързан с базата данни, която използва.

Друго голямо предимство на Web-програмирането е неговата разпределеност. Поради разпределеността на WWW, разработката на разпределени Web-приложения е доста лесно. Достатъчно е приложението да се раздели на компоненти и всеки от тях да се разположи на различен Web-сървър. Така отделните компоненти, въпреки че са физически разделени, могат да работят като една цяла система. Използвайки съвременни технологии за балансиране на натоварването и възстановяване от грешки, надеждността на Web-приложенията става изключително голяма, като това не коства много допълнителни усилия за програмиста.

Несесийност на HTTP протокола

По принцип повечето Web-приложения са предназначени за многопотребителски достъп. Използването на протокола HTTP в Web-програмирането създава проблеми в тази насока, защото има несесиен характер, т.е. няма вградена възможност за идентификация и проследяване на потребителските сесии. Първоначално HTTP е служел за достъп до статични ресурси (HTML файлове, изображения, др.) и за тази си функция е бил чудесен. В днешно време HTTP се използва за много други неща, които не са били предвидени при създаването му и за които не е много удобен, но се използва, защото се е наложил като стандарт. Въпреки че версия 1.1 на HTTP протокола въвежда така наречените keep-alive връзки (възможност за изпълнение на няколко HTTP заявки през веднъж осъществена връзка между клиента и сървъра), това не решава проблема с еднозначната идентификация на потребителя от страна сървърската част на Web-приложението. За щастие в J2EE средствата за осигуряване на многопотребителски достъп се дават стандартно от платформата, поради което за да е възможно няколко потребителя да работят едновременно и независимо един от друг с едно Web-приложение, от програмиста не се изискват никакви допълнителни усилия. Достатъчно е да се използват съответните стандартни обекти за работа с потребителска сесия, които се дават от контейнера за приложения.

Производителност

Производителността е един от големите плюсове на Web-програмирането. При този вид програмиране бизнес операциите на системата са изнесени на сървъва, като по този начин ролята на клиента се свежда до това да обработва и визуализира HTML документи. И тъй като HTML е сравнително прост описателен език, изискванията и натоварването на клиентската машина са минимални.

Сигурност

Трябва да отбележим несъмненото превъзходство на Web-програмирането пред конвенционалното програмиране по отношение на сигурността.

От страна на клиента

В компютърната индустрия вирусите, които се разпространяват с изпълнимите програми, са нанесли щети на много хора и корпорации. Използването на антивирусен софтуер не винаги спасява потребителя от действието на злонамерени програми, а освен това забавя бързодействието на компютъра му. За разлика от настолният софтуер, Web-приложенията са напълно безопасни за

потребителя, тъй като при тях клиентската част се състои във визуализирането на документи, описани с HTML/CSS, и изпълнението на JavaScript.

От страна на сървъра

Предимството на Web-приложенията по отношение на сигурността се проявява най-вече от гледна точка на сървъра. Голям брой клиенти могат да имат достъп до някакъв ресурс на сървъра (база данни, изчислителна мощ, т.н.), без да могат да го достъпват директно, а само посредством някое Web-приложение. Съответното Web-приложение може да дава достъп до определени ресурсри само след автентикация на достъпа с парола, клиентски сертификат или друга технология. Въвеждането на HTTPS като стандарт прави невъзможно подслушването на конфиденциална информация и дава голяма сигурност на Web-приложенията.

Неодстатъци на Web-програмирането

Първият недостатък на Web-приложенията се отнася до производителността. Бавната връзка между клиента и сървъра води до ниска производителност на цялата система. Макар и сървърът да обработва и отговаря мигновенно на клиентските заявки, потребителите често имат усещането, че системата, с която работят, е бавна заради забявянето, което е необходимо за пренос на данните между сървъра и клиента. Заради това забавяне Web-приложенията трябва да се разработват внимателно, особено ако се очакват потребители с лоша Интернет връзка.

Другият проблем за Web-програмирането е статичният характер на HTML. Въпреки че JavaScript донякъде решава този проблем, разработката на сложен, интерактивен и удобен потребителски интерфейс често пъти е много трудна задача, която изисква сериозни усилия и творчество. При силно интерактивни Web-приложения обикновено има проблеми с различните Web-браузъри и различните версии на един и същ браузър. С HTML в комбинация с CSS и JavaScript не може да се направи всичко, дори ако програмиста показва завидно майсторство. Понякога изискванията на крайния потребител относно потребителския интерфейс се указват просто неосъществими и се налага да бъдат променяни, за да стане възможно изпълнението им. Поради факта, че Web-приложенията не са подходящи за създаване на сложен потребителски интерфейс, обикновено интерфейсът на Web-базирания софтуер е прост, макар и достатъчно функционален.

Друг, много сериозен проблем на Web-програмирането е еднопосочната връзка между потребителя и клиента. Web-програмирането е базирано на механизма "заявка-отговор", който непозволява на сървъра да изпраща данни на клиента без негова заявка. Това силно затруднява някои интерактивни приложения, които разчитат на получаване на данни асинхронно от сървъра. Пример за такива приложения са приложенията за разговори (chat), които често пъти се реализират с аплети, поради слабостта на HTML, CSS и JavaScript. Докато не се намери добро решение на проблема с еднопосочността на комуникацията, Web-приложенията никога няма да достигнат функционалността на настолните приложения.

Защо Web-програмирането е толкова разпространено

Въпреки всичките изложени недостатъци, Web-програмирането си остава един от най-предпочитаните подходи за разработка на големи многопотребителски приложения. Най-важната причина за това големите компании да предпочитат Web-базирани приложения, е че поддръжката на Web-приложенията е значително по-лесна, отколкото на настолните. Преминаването към нова версия на едно Web-приложение става без инсталиране на нищо допълнително. Просто се подменя сървърската част на приложението и всички потребители (които понякога може да са милиони, дори десетки милиони) не трябва да правят абсолютно нищо, за да преминат на новата версия. Достатъчно е да заредят отново адреса на Web-приложението от своя Web-браузър и започват работа с новата система.

Java базирани Web-приложения

Едно Java базирано Web-приложение представлява съвкупност от сървлети, JSP-та (разширение на сървлетите, което позволява в HTML документи да се вгражда Java код), Java архиви и други файлове, които заедно изграждат една обща Web-система. Файловете на приложението се разполагат в поддиректориите на една директория с фиксирана структура, която се задава от J2EE спцификацията. JSP файловете се разполагат в главната директория на приложението. Настройките на приложението се задават в специален файл с име web.xml намиращ се в поддиректория WEB-INF. Класовете, които приложението използва се разполагат в поддиректория WEB-INF\classes. Java архивите, които могат да съдържат класове, изображения и други ресурси се разполагат в поддиректория WEB-INF\lib. Така сървърът, който изпълнява приложението (Web-контейнерът) знае къде да търси различните файлове, когато му потрябват.

Java Сървлети

Java платформата дава няколко стандартни средства за създаване на динамични Web-страници. Основната технология, на базата на която се изгражда всичко останало са ***сървлетите***. Сървлетите представляват програми на Java, които приемат като вход някакви данни от потребителя, обработват ги и връщат като резултат динамично генериран HTML или друг документ. Например, един сървлет може да приема като входни данни име на потребител и парола, да проверява валидността им по някакъв начин и да пренасочва браузъра към друга страница от Web-приложението, ако са валидни или да връща съобщение за грешка в противен случай.

Предимства на сървлетите

От гледна точка на ефективността сървлетите превъзхождат стандартната CGI технология, защото изпълнението на сървлет не води до създаване на нов процес в операционната система, което е традиционно бавна операция. Java виртуалната машина стои постоянно заредена в паметта и когато се извика някой сървлет, той просто се изпълнява и резултатът се връща на клиента, без да се създава нов процес за обработка на заявката. Освен това, веднъж изпълнен, сървлетът остава в компилиран вид активен в паметта, чакайки ново извикване. Това допълнително повишава производителността. По подобен начин работят повечето съвременни Web-технологии като PHP, ASP и Perl. При тези технологии обикновено интерпретаторът на съответния скрипт език стои като модул зареден постоянно в паметта на Web-сървъра и се включва при клиентска заявка. Така производителността зависи от производителността на интерпретатора. Заради традиционната не много добра производителност на Java виртуалната машина, може да се спори дали JSP/сървлет технологията е по-бърза или по-бавна от останалите, но при всички случаи скоростта ѝ не се различава значително от тази на конкурентните технологии.

Съвкупността от средства (framework-ът) за работа със сървлети и JSP-та, който ни дава J2EE платформата предлага доста вградени удобства. Едно от тях е вече споменатото автоматично управление на многопотребителския достъп, което дава възможност за съхранение на различни данни за потребителя в рамките на неговата сесия. Освен това извличането и декодирането на параметрите е силно улеснено, дава се възможност за достъп на по-високо ниво до HTTP хедърите, за пренасочване на потребителския браузър, за достъп до общи за приложението данни, за обмяна на данни между приложенията, както и достъп до ресурси, принадлежащи на Web-контейнера (сървърът, изпълняващ съврлетите и JSP-тата).

Java базираните Web-приложения са изключително лесно преносими. В повечето случаи всичко, което е необходимо за да се прехвърли едно приложение от един сървър на друг, е да се прехвърли един единствен файл. Дори новият сървър да е от друг производител и да работи на друга операционна система, рядко се налага да се извършват промени или допълнителни настройки за да заработи Web-приложението на новия сървър. Тази изключително добра

съвместимост се дължи на стандартите за Web-приложения, които се дават от платформата J2EE и се спазват стриктно от почти всички производители на Web-приложения и Web-контейнери.

Сигурността и надеждността на Java базираните Web-приложения е изключително голяма. Това се дължи на надеждността и сигурността на самия език Java. Уязвимости като препълване на буфери са изключени, а проблеми с лошо декодиране на данни, неправилна работа с базата данни, непозволен достъп до паметта и още много други при Java платформата са значително по-трудно осъществими, отколкото при другите платформи.

Друго важно предимство на Java сървлетите, JSP-тата и Java-базираните Web-приложения е че те могат да се използват напълно безплатно. За работата им не е необходимо закупуването на скъп сървърски софтуер, защото има достатъчно добри безплатни Web-контейнери и J2EE сървъри за приложения (application servers), които са идеални за малкия и средния бизнес. Пример за такива безплатни сървъри са Web-контейнерът Tomcat (http://jakarta.apache.org), който ще разгледаме след малко и сървърът JBoss (http://www.jboss.org), който е почти пълна имплементация на J2EE платформата.

Структура на сървлетите

За създаването на Java сървлет е необходимо да се наследи класа `javax.servlet.http.HttpServlet` и да се припокрие метода `doGet()` или `doPost()` ако сървлетът ще обработва съответно GET или POST HTTP заявки. И двата метода doGet() и doPost() приемат като аргументи два обекта `HttpServletRequest` и `HttpServletResponse`. HttpServletRequest служи за извличане на входните параметри на сървлета – данните от HTML форми, изпратени от потребителския Web-браузър, хедърите на HTTP заявката, информация за клиента, неговия IP адрес и браузър. HttpServletResponse служи за изпращане на данни в отговор на получената HTTP заявка. Позволява задаване на кода на резултата и полетата в хедъра на HTTP отговора, както и разбира се, самия текст на отговора. При простите сървлети по-голямата част от сървлета представлява код, който отпечатва динамично създадения HTML документ в изходния поток на заявката. При по-сложните сървлети се налага работа с HTTP хедърите, работа с cookies и други, които ще разгледаме по-късно. Не всички сървлетите са HTTP сървлети. Има сървлети, които обслужват други протоколи и за реализацията им се наследява класа `GenericServlet`, а не `HttpServlet`. В нашия курс ще бъдат разглеждани само HTTP сървлети. Да дадем пример за прост HTTP сървлет:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class NakovFirstServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
    throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("The time is: " +
            new java.util.Date());
        out.println("</HTML>");
    }
}
```

Всичко, което прави дадения сървлет `NakovFirstServlet` е да наследи класа `HttpServlet` и в метода `doGet()` да получи изходния поток на отговора на заявката и да отпечата в него съвсем прост HTML документ, който се състои от 3 реда и съдържа текущата дата и час.

Работа със сървъра Tomcat

Сървърът Tomcat представлява безплатен Web-контейнер, който може да изпълнява Java сървлети, JSP-та и Web-приложения. Tomcat е Web-сървър, написан на Java, който освен статични

файлове, може да връща и динамични документи, създадени в резултат от изпълнението на сървлет или JSP.

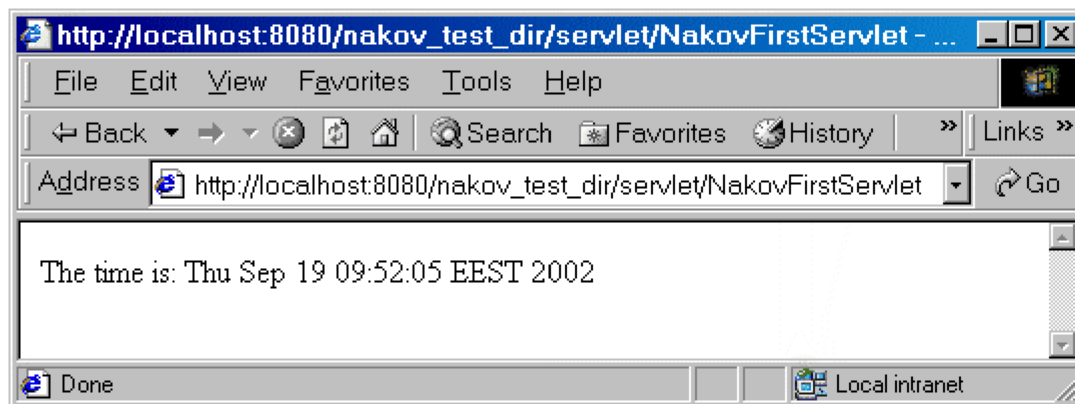
Инсталиране на Tomcat

Сървърът Tomcat е достъпен от адрес http://jakarta.apache.org/. След като издърпате последната версия, която представлява един .zip файл, го трябва да го разархивирате в някоя директория. Препоръчва се в името на директорията да не се съдържат интервали, защото интервалите служат за разделители в Java и могат да създадат някои досадни проблеми. Ако не сте инсталирали JDK на вашия компютър, инсталирайте последната версия. Можете да я намерите на адрес http://java.sun.com/j2se/. Добавете в променливите на средата променливата `JAVA_HOME` със стойност директорията където е инсталирана JDK. Това може да стане например от конзолата с командата "`set JAVA_HOME=C:\jdk1.3.1`". Намерете директорията, в която сте инсталирали Tomcat. Да предположим, че това е `C:\Tomcat`. В поддиректория `bin` има файл за стартиране `startup.bat` (`startup.sh` за Unix/Linux), с който можете да стартирате сървъра. Стартирайте този файл. Това е всичко необходимо за стартиране на сървъра. За да проверите дали работи, стартирайте вашия Web-браузър и въведете адреса http://localhost:8080/. Ако всичко е наред, ще се появи заглавната страница на Tomcat. Можете да тествате и стандартните примерни сървлети и JSP-та, като следвате линковете от главната страница. Забележете, че по подразбиране Tomcat работи на порт 8080, а не на стандартния за протокола HTTP порт 80.

Как да стартираме нашия сървлет

Освен поддиректорията `bin`, в директорията на Tomcat има още една важна поддиректория – `webapps`. В тази поддиректория стоят всички инсталирани Web-приложения. Директория `webapps\ROOT` е главната виртуална директория на сървъра. За нашите тестови цели трябва да създадем нова поддиректория в `webapps`. Един полезен съвет при разучаване на непознат сървър е когато задавате имена на нови файлове и директории, имена на проекти, имена на услуги и т.н. да избирате нестандартно име, за да можете да различите след това стандартните неща от нещата, които вие сте създали. По този принцип не трябва да кръщавате новата директория за вашите тестове с имена като `test`, `samples`, `examples`, `tests`, `web`, `root` и т.н. защото рискувате първо избраното от вас име да съвпадне с някое служебно име със специално предназначение и второ ще ви е трудно да различите стандартните имена, идващи по подразбиране със сървъра, който разучавате, от вашите, които вие сте създали. Аз лично в такива случаи обикновено задавам за име "`nakov_directory`", "`nakov_service`" или нещо подобно, което ми подсказва че това име е избрано от мен и какво точно се крие зад това име – директория, услуга или нещо друго. Да предположим, че сме създали директорията "`nakov_test_dir`". За да изпълним нашия сървлет е необходимо да създадем поддиректория на новосъздадената с име `WEB-INF\classes`, да копираме некомпилирания сървлет в нея и да го компилираме. Ако сме инсталирали Tomcat в `C:\Tomcat`, а за нашиите тестове сме избрали директория `nakov_test_dir`, ще трябва да запишем нашия тестов сървлет `NakovFirstServlet.java` в директория `C:\Tomcat\webapps\nakov_test_dir\WEB_INF\classes`. За да го компилираме ще имаме нужда от пакета `javax.servlet.*`, защото той не се разпространява стандартно с JDK. Класовете от този пакет са включени в Tomcat и се намират във файла `C:\Tomcat\lib\servlet.jar`. След като компилираме нашия сървлет и получим файла `NakovFirstServlet.class` в директорията `C:\Tomcat\webapps\nakov_test_dir\WEB_INF\classes`, за да го изпълним, е необходимо да стартираме сървъра чрез скрипта `startup.bat` от `bin` директорията на Tomcat. По подразбиране в Tomcat всички файлове от главната директория на едно приложение се публикуват във виртуална директория с име името на приложението, а всички `.class` файлове от `WEB-INF\classes` директорията се публикуват като сървлети във виртуална директория с име `<име_на_приложението>/servlet`. Изключение прави специалната поддиректория `WEB-INF`, която остава недостъпна през браузъра. Така нашият сървлет е достъпен автоматично от адрес `http://localhost:8080/nakov_test_dir/servlet/NakovFirstServlet`, а всички файлове от директория `C:\Tomcat\webapps\nakov_test_dir\` – от адрес `http://localhost:8080/nakov_test_dir/`. Имайте предвид, че ако промените файловете от нашата тестова директория, промените ще са видими

веднага, но ако промените и прекомпилирате тестовия сървлет, е необходимо да рестартирате Tomcat, за да влязат в сила промените. Това се обяснява с кеширането на заредените класове, което Tomcat прави с цел по-голяма производителност. Можете да тествате първия си сървлет с вашия Web-браузър. Ще получите нещо подобно на това:



В предходната част от курса се запознахме с основните концепции в Web-програмирането, обяснихме какво е Java сървлет, как се създават сървлети и как се изпълняват със сървър Tomcat. В тази част ще продължим изучаването на Java сървлетите, ще покажем как сървлетите могат да приемат данни въведени от потребителя във вид на параметри, ще обясним жизнения цикъл на сървлетите и как те могат да работят с много потребители едновременно като използват HTTP сесии.

HTML форми и извличане на данните от тях

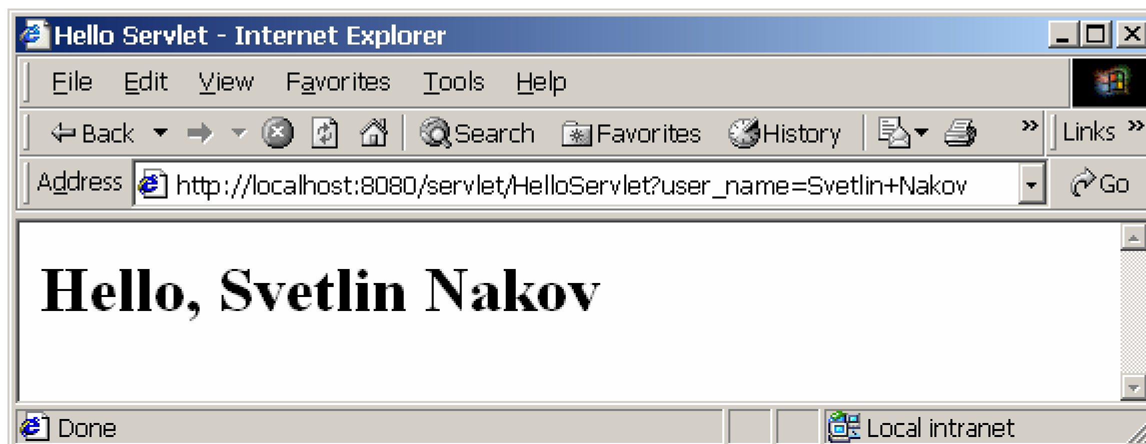
Всички сме използвали машини за търсене в Интернет като Google и AltaVista и знаем, че те представляват Web-приложения, които приемат от потребителя няколко ключови думи и му намират страниците, в които тези думи се срещат. Вероятно всеки е забелязал че след задаване на заявката за търсене в полето за адрес на браузъра се появява ново URL съдържащо въведената фраза за търсене замаскирана сред множество странни символи. Например ако в Google зададем търсене на фразата “Svetlin Nakov”, ще получим URL подобно на това: http://www.google.com/search?q=Svetlin+Nakov&ie=windows-1251&hl=bg&lr=. Частта от URL-то след въпросителния знак съдържа данните, изпратени като параметри към това URL, кодирани по специален начин, наречен URL-encoding. Данните от HTML форма могат да бъдат предадени към сървъра по два начина – с GET или POST заявка. При GET HTTP заявки, те се предават след URL-то като се отделят от него с въпросителен знак, а при POST HTTP заявки се предават заедно със заявката, отделени от URL-то на отделен ред.

Java сървлетите имат вградена възможност за извличане на изпратените от потребителя данни. За целта се използва методът getParameter() на класа HttpServletRequest. Този метод връща стойността на параметър по зададено име или null ако такъв параметър не е изпратен от браузъра на потребителя. Парсването на параметрите и декодирането им от URL-encoded формат в чист текст става напълно автоматично, т.е. програмистът не е необходимо да се грижи за отделянето на параметрите един от друг, за отделянето им от URL-то и за декодирането на символите, които са били заменени с други съгласно с цел да се избегнат (escaped symbols). Ето един примерен сървлет, който демонстрира леснотата с която се получават параметрите. Той получава като вход име на потребител (параметър с име user_name) и му казва “здравей”.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {
    public void doGet (HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
        resp.setContentType("text/html");
        ServletOutputStream out = resp.getOutputStream();
        String userName = req.getParameter("user_name");
```

```
            out.println("<HTML>");
            out.println("<HEAD><TITLE>Hello Servlet</TITLE></HEAD>");
            out.println("<BODY>");
            out.println("<H1>Hello, " + userName + "</H1>");
            out.println("</BODY></HTML>");
        }
}
```

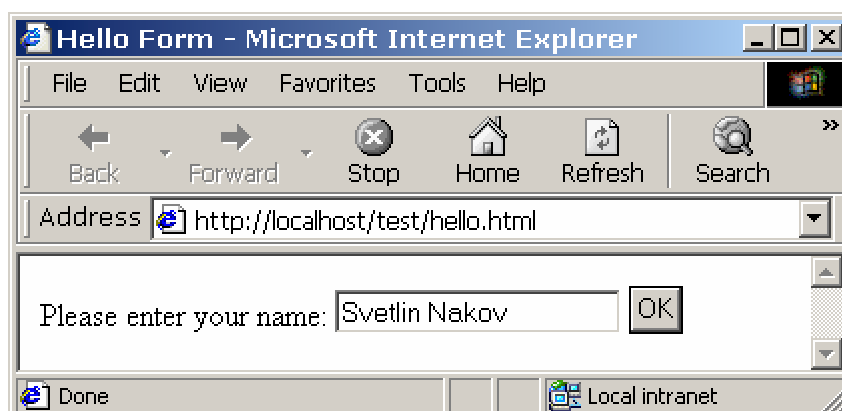
Ето и резултатът от изпълнението на този сървлет с параметър “Svetlin Nakov” :



Можем да направим HTML форма, в която потребителя си пише името и след натискане на “submit” бутона това име се подава като параметър на сървлета HelloServlet. Ето един пример:

```
<html>
<head> <title> Hello Form </title> </head>
<body>

<form method="GET" action="/servlet/HelloServlet">
Please enter your name:
<input type="text" name="user_name">
<input type="submit" value="OK">
</form>

</body></html>
```

Примерната HTML форма задава за метод на HTTP зявката GET. Така при натискане на “submit” бутона въведения в полето user_name текст се предава като долепен до URL-то параметър с име “user_name” и стойност въведената в текстовото поле. Ето как изглежда формата в браузъра:



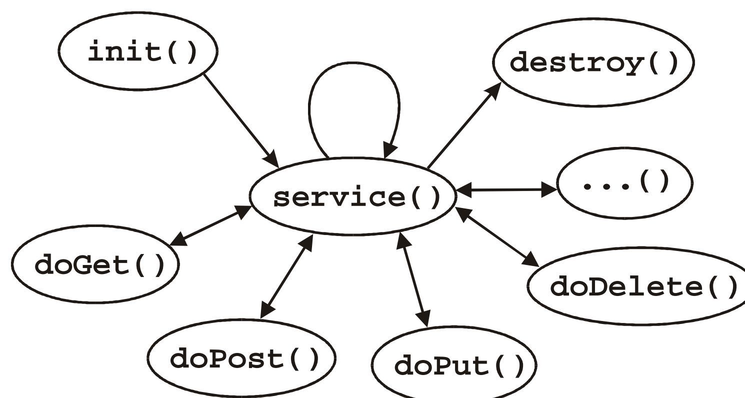
По идея HTML формите служат за автоматизация на процеса на предаване на параметри между потребителския браузър към сървърски скриптове, които обработват тези параметри. Името на скрипта, който браузърът извиква при submit-ване на формата, се задава в атрибута “action”, а методът на HTTP заявката – в атрибута “method” на тага <form>. Във формата се задават различни текстови и други полета, като им се задават имена. Зададените имена съвпадат с имената на параметрите, които се генерират при създаване на HTTP заявката към сървъра. Специалният бутон “submit” служи за изпращане на данните, въведени във формата, към посочения скрипт. При изпращането на попълнените във формата данни браузерът се грижи да ги кодира по стандарта “URL-encoding” и да ги изпрати през URL-то или като част от заявката в зависимост дали методър

на формата е GET или POST. При използването на GET метод всички параметри се долепват в URL-то, поради което обемът им не може да бъде много голям. При използването на POST метод всички параметри се предават скрито, не през URL-то и потребителят вижда само името на скрипва, който е обработил данните, но не и самите данни. Кой от двата метода да се използва е въпрос на преценка от страна на Web-разработчика.

За удобство на програмиста класът `HttpServletRequest` има и метод за изброяване на всички изпратени параметри `getParameterNames()`. Трябва да се знае, че при имената на параметрите малките и главните букви се различават.

Жизнен цикъл на сървлетите

Жизненият цикъл на сървлетите описва тяхното поведение от момента, в който те бъдат създадени като обекти на сървъра (инстанцирани) до момента на тяхното премахване от него. На картинката са показани основните методи, който реализират жизнения цикъл на сървлетите и последователността на тяхното извикване:



При първо извикване на сървлета сървърът, който изпълнява сървлетите и JSP скриптовете (така наречения Web-контейнер), извиква метода init(), дефиниран в класа HttpServet. Сървлетите, които имат нужда от еднократна първоначална инициализация преди започване на работата си, трябва да припокрият този метод и да реализират тази своя инициализация. Например един сървлет може да прочете от сървъра или от някакъв файл конфигурационна информация, която да използва по-нататък.

При настъпване на заявка за достъп до сървлета, подадена от Web-браузъра на някой клиент, сървърът извиква метода service() от класа HttpServlet. Този метод анализира типа на заявката и взависимост от това дали заявката е GET, POST, PUT, DELETE или друга, извиква съответно един от методите doGet(), doPost(), doPut(), doDelete() и т.н. Понеже HTTP методите PUT, DELETE, HEAD, TRACE, OPTIONS и т.н. се използват много рядко и не са типични за повечето сървлети, няма да ги разглеждаме. Методът doGet(), трябва да бъде реализиран от сървлетите, които обработват заявки подадени по метод GET, например ако обработват данните, получени от HTML форми, за които е зададено method="GET". Аналогично doPost() методът трябва да бъде реализиран, когато трябва да се обработят данни получени от HTML форми, за които методът е POST. Ако искаме да направим сървлет, който обработва едновременно и GET и POST заявки, не е хубаво да припокриваме директно метода service(), защото той се грижи за правилната обработка на HEAD заявките и за още други важни неща. Вместо това можем да имплементиране обработката на данните в метода doGet(), а от doPost() просто да извикваме doGet(). Това е препоръчителният начин за обработка на данни, които се очаква да пристигат и чрез GET и чрез POST заявки.

След като сървлетът бъде изпълнен веднъж, той остава като обект в паметта на виртуалната машина на Java и при следващи извиквания се изпълнява веднага, без да се зарежда от .class файла отново. В рамките на едно Web-приложение един сървлет се инстанцира само веднъж. Когато няколко клиента поискат един сървлет едновременно, Web-контейнерът стартира едновременно няколко нишки (threads) и извиква от всяка от тях сървлета в един и същ момент. Понеже всеки сървлет има само една инстанция в сървъра, то класът, който реализира този сървлет, заедно с

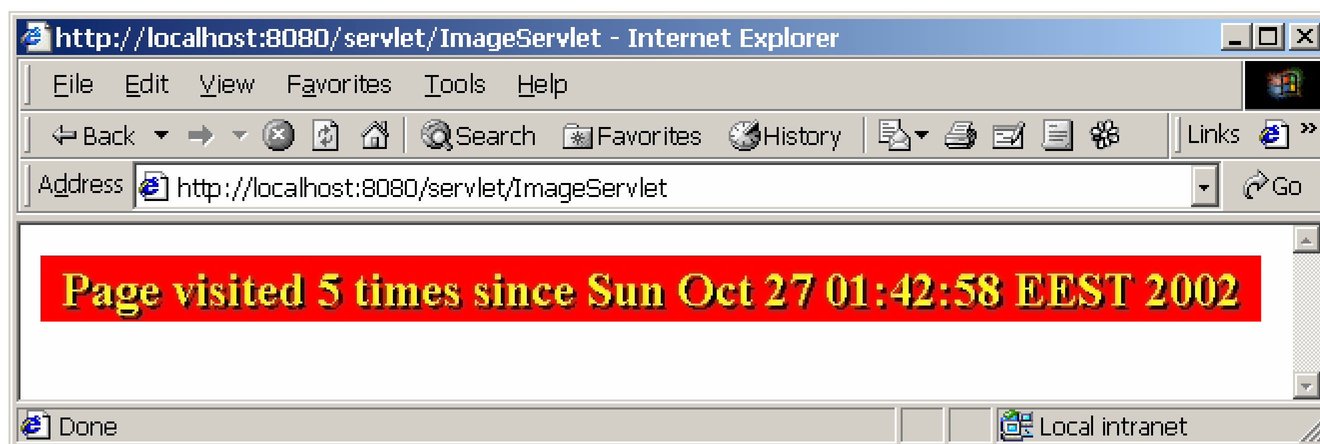
член-променливите, които са дефинирани в него, се инстанцират само веднъж в рамките на Web-приложението. Следователно работата с тези член-променливи не е thread-safe, т.е. не е обезопасена от проблеми с конкурентния достъп при заявки от няколко потребителя едновременно. Затова е необходимо програмистът да има предвид, че е възможно кодът на единствената инстанция на написания от него сървлет да се изпълнява едновременно от няколко нишки (threads) и затова трябва да се грижи за синхронизация на достъпа до член-променливите на сървлета, както и другите ресурси, които използва.

Когато сървърът по някаква причина реши да премахне от паметта един сървлет (например при намеса на администратора), се извиква метода destroy() на класа HttpServet. В реализацията на този метод сървлетите трябва да освободят заетите от тях ресурси и да финализират работата си. Типичен пример за използване на init() и destroy() методите при прости приложения е за отваряне и затваряне на връзката към базата данни, когато се използва такава. При по-сложни приложения връзката към базата данни се управлява от специална компонента на системата известна като “connection pool”. Методът destroy() се използва по-рядко от метода init(), защото Java освобождава автоматично някои типове ресурси, като например паметта, при унищожаването на обект. Ето един пример за сървлет, който използва методите init(), destroy(), doGet() и doPost() и същевременно демонстрира как от сървлет може да се генерира динамично JPEG изображение и да се върне като отговор на клиентската заявка:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.*;
import java.awt.image.*;
import com.sun.image.codec.jpeg.*;
import java.io.*;
import java.util.Date;

public class ImageServlet extends HttpServlet {
    private int mVisitCounter;
    private String mStartDate;

    public void init() {
        mStartDate = new Date().toString();
        mVisitCounter = 0;
    }

    public BufferedImage createImage(String msg) {
        Font font = new Font("Serif", Font.BOLD, 24);
        FontMetrics fm =
            new Canvas().getFontMetrics(font);
        int width = fm.stringWidth(msg) + 20;
        int height = fm.getHeight();
        BufferedImage image = new BufferedImage(
            width, height, BufferedImage.TYPE_INT_RGB);
        Graphics g = image.getGraphics();
        g.setColor(Color.red);
        g.fillRect(0, 0, width, height);
        g.setFont(font);
        g.setColor(Color.black);
        g.drawString(msg, 12, fm.getAscent()+2);
        g.setColor(Color.yellow);
        g.drawString(msg, 10, fm.getAscent());
        return image;
    }

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
    throws IOException, ServletException {
        String msg;
        synchronized(mStartDate) {
            mVisitCounter++;
            msg = "Page visited " + mVisitCounter +
                " times since " + mStartDate;
        }
        BufferedImage image = createImage(msg);
        response.setContentType("image/jpeg");
        OutputStream out = response.getOutputStream();
```

```
        JPEGImageEncoder encoder =
            JPEGCodec.createJPEGEncoder(out);
        JPEGEncodeParam jpegParams =
            encoder.getDefaultJPEGEncodeParam(image);
        jpegParams.setQuality(1, false);
        encoder.setJPEGEncodeParam(jpegParams);
        encoder.encode(image);
        out.close();
    }

    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
    throws IOException, ServletException {
        doGet(request, response);
    }
}
```

При инициализация, в метода init(), сървлетът запомня във вътрешна член-променлива датата и часа, в който е инициализиран. В друга вътрешна член-променлива той помни и броя пъти, които е бил извикван чрез GET или POST заявка. При извикване на doGet() метода, сървлетът генерира текстово съобщение, което съобщава колко пъти е била посетена страницата от първото извикване на сървлета. Заради възможността няколко потребителя едновременно да поискат страницата, е необходимо достъпът до член-променливите на сървлета да бъде синхронизиран. Както знаем от темата за многонишково програмиране и синхронизация, в Java синхронизацията може да се прави по монитора на произволен обект. В нашия случай синхронизираме по обекта, съдържащ датата и часа на инициализация на сървлета. Понеже, както споменахме по-горе, сървлетите се инстанцират само веднъж в рамките на едно Web-приложение, член-променливите им също се инстанцират само веднъж и затова броячът на посетители има само едно копие в паметта, въпреки, че не е обявен като static. Благодарение на синхронизирания достъп до него, той отчита посетителите правилно, дори при конкуретно извикване от много потребители едновременно. За динамичното генериране на изображението се използват стандартните средства на jawa.awt. Изчисляват се размерите на текста при шрифта, който ще бъде използван, след това се създава изображение от класа BufferedImage, взема се неговия обект за графична манипулация Graphics и се изобразява текста. Първо изображението се запълва с червен цвят, след това се печата текста, малко отместен с черен цвят, а след това се печата с жълто същия текст на истинската му позиция. Така се получава текст със сянка. След като изображението е изготвено, то се конвертира в JPEG поток от данни като се използва кодекът на Sun за JPEG кодиране и му се задава да работи с максимално ниво на качество. Полученият поток от данни се изпраща като изход от сървлета. За да може Web-браузърът да разпознае изпратения поток от данни като JPEG картинка, а не като текст, в header-а на HTTP отговора на заявката се слага "Content-type: image/jpeg". По подразбиране, ако сървлетът не укаже друго, за тип на изпратените данни се слага "Content-type: text/html". Реализацията на метода doPost() просто извиква метода doGet(), което позволява на сървлета да отговаря както на GET, така и на POST заявки по протокола HTTP. Ето и примерен резултат от извикването на сървлета:



Поддръжка на потребителски сесии

Когато говорихме за Web-приложения, споменахме, че те имат възможност да обслужват едновременно много потребители, независимо един от друг. Как знаем, HTTP протоколът има несесиен характер, т.е. не ни предоставя възможност да различаваме потребителите един от друг и да проследяваме коя заявка от кой потребител идва. Ето защо за проследяване и разграничаване на потребителите един от друг са необходими допълнителни усилия, които Web-приложенията трябва да полагат. Потребителска сесия наричаме периода, в който един потребител си взаимодейства с едно Web-приложение. Проследяването на последователността от заявки, извършени от един потребител се нарича проследяване на неговата сесия. Ако два потребителя работат едновременно с една Web-система, те имат две различни сесии. Пример за Web-приложение, което проследява потребителската сесия, е Web-базираната система за електронна поща на Yahoo. Всички знаем, че е възможно докато един потребител си чете пощата от mail.yahoo.com, друг потребител, напълно независимо от него също да си чете пощата от същия сайт. Web-приложението за електронна поща, работещо на машината с име mail.yahoo.com разпознава различните потребители и проследява техните сесии. Взависимост от това кой потребител е дал HTTP заявка към Web-приложението, сървърът разпознава неговата сесия и дава достъп до неговите email-и, а не до тези на останалите потребители, работещи в същия момент. Възможно е от един и същ компютър да се осъществят няколко независими сесии към едно и също Web-приложение. Например потребителят може да отвори два различни Web-браузъра – един Netscape и един Opera и да влезе в едно Web-приложение като два различни потребителя. В рамките на браузъра Netscape, той ще има създадена една сесия със сървъра, а в рамките на браузъра Opera той ще има създадена още една, независима от първата сесия със същия сървър. Това означава, че за сървъра потребителите са различими един от друг, дори когато идват от един и същ компютър. Това се обяснява с механизма, по който сървърът различава потребителите един от друг. Има два основни начина за проследяване на потребителската сесия – с cookies и с добавяне на допълнителен параметър към URL-то. Cookies е възможност едно Web-приложение да чете и записва информация на машината на клиента. Информацията от cookies може да се чете само от приложението, което я е записало и може да изчезва ако не се ползва дълго време, взависимост от параметрите зададени при създаването ѝ. Посредством cookies Web-приложенията могат да записват на машината на потребителя някакъв идентификатор на сесия и след това като я прочитат при всяка заявка, да разпознават потребителя. Другият начин за следене на потребителските сесии е чрез добавяне на допълнителен параметър към URL-то. При започване на работа на потребителя се генерира уникален ключ и той се добавя като параметър при всяка GET или POST заявка. Този подход изисква допълнителни усилия за добавяне на скрити полета във всяка HTML форма и добавяне на параметри към всеки hyperlink и затова се използва рядко, обикновено когато клиентският браузър не поддържа cookies или потребителят ги е забранил. С cookies усилията за проследяване на потребителите са значително по-малки.

В Java сървлетите и JSP скриптовете поддръжката на потребителски сесии е напълно автоматична. Програмистът не е нужно да изпраща и чете cookies или да добавя и разпознава след това допълнителни параметри към URL-то. Достатъчно е да се използва API-то за работа със сесии, което се дава от framework-а за Web-приложения. Основен е класът HttpSession, който представя потребителската сесия. Получаването на обект, асоцииран с текущата сесия можем да вземем от HttpRequest обекта по следния начин:

```
HttpSession session = request.getSession();
```

Ако сесия не съществува такава ще бъде създадена. Взимането на сесията за един и същ потребител връща един и същ обект, а за различни потребители връща различни обекти. За всеки нов потребител се създава нов обект от класа HttpSession и се връща този обект. В обекта session могат да се съхраняват произволни данни за потребителя посредством методите cетAttribute(key, value) и getAttribute(key). Веднъж съхранени в сесията, тези обекти са достъпни по време на всяка заявка от потребителя в рамките на тази сесия. Ето един типичен пример за използване на сесия е когато на всеки потребител се предоставят различни ресурси в зависимост от това като какъв се е автентикирал, а неавтентикираните потребители не се допускат. Представяме си сорсовете на сървлета LoginServlet:

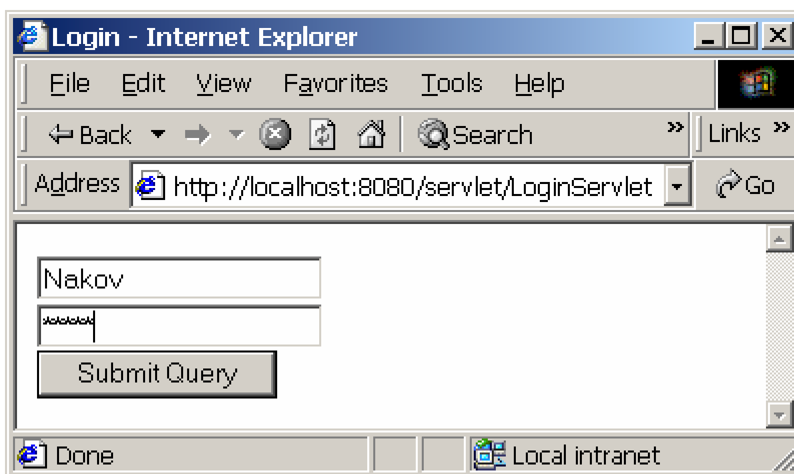
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LoginServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
    throws IOException, ServletException {
        String user =
            request.getParameter("user");
        String password =
            request.getParameter("password");
        PrintWriter out = response.getWriter();
        if ((user==null) || (password==null)) {
            showLoginForm("", out);
        } else if (user.equals(password)) {
            HttpSession session =
                request.getSession();
            session.setAttribute("USER", user);
            response.sendRedirect(
                "/servlet/MainServlet");
        } else {
            showLoginForm(
                "Invalid login.<br>", out);
        }
    }

    private void showLoginForm(
        String captionText, PrintWriter out)
    {
        out.println(
            "<html><title>Login</title>\n" +
            "<form method=\"GET\" action=" +
            "\"/servlet/LoginServlet\">\n" +
            captionText +
            "<input type=\"text\"" +
            " name=\"user\"><br>\n" +
            "<input type=\"text\"" +
            " name=\"password\"><br>\n" +
            "<input type=\"submit\">\n" +
            "</form></html>"
        );
    }
}
```

и сървлета MainServlet:

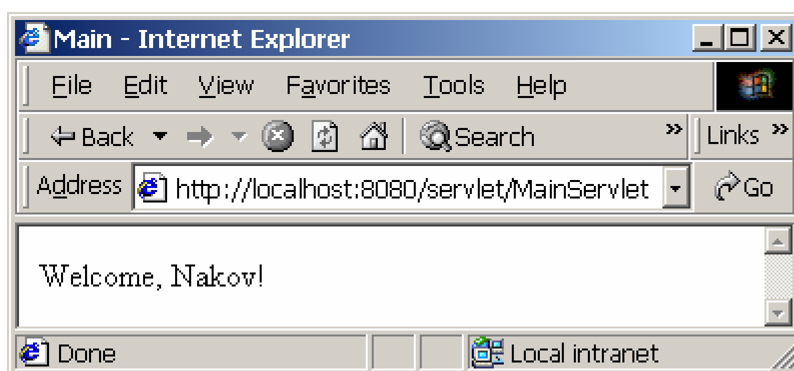
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MainServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
    throws IOException, ServletException {
        HttpSession session =
            request.getSession();
        String user = (String)
            session.getAttribute("USER");
        PrintWriter out = response.getWriter();
        if (user==null) {
            showMainForm("Not authenticated." +
                " Please <a href=\""+
                "/servlet/LoginServlet\">" +
                "login</a> first.", out);
        } else {
            showMainForm(
                "Welcome, " + user + "!", out);
        }

    }
```

```
    private void showMainForm(
        String captionText, PrintWriter out)
    {
        out.println(
            "<html><title>Main</title>\n" +
            captionText + "</html>"
        );
    }
}
```

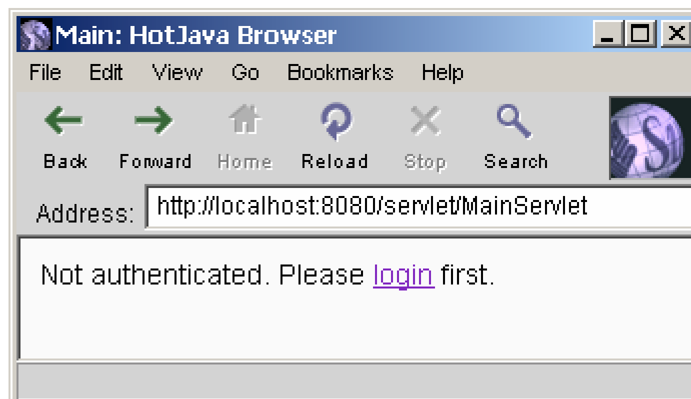
Първият сървлет (LoginServlet) служи за автентикация на потребителите. Когато се извика без параметри, той показва HTML форма за попълване на име на потребител и парола:



Когато се попълни и изпрати формата, се извиква същият сървлет, но въведените име на потребител и парола се изпращат към него като параметри. Когато сървлетът разпознае валидна комбинация от потребителско име и парола, записва в автоматично създадената за текущия потребител сесия под ключ “USER” въведеното потребителско име. За простота в нашия пример валидни комбинации от потребителско име и парола са всички, в които потребителското име съвпада с паролата. След успешна автентикация потребителят се препраща към основния сървлет (MainServlet) чрез response.sendRedirect(URL). Основният сървлет разпознава неговата сесия по съществуването стойност под ключа “USER”, прочита от нея името на автентикирания потребител и го поздравява с кратко съобщение:



Ако в същия момент друг потребител се опита да зареди MainServlet-а, като напише адреса му в браузъра си, той ще бъде разпознат от системата като различен от първия. В неговия HttpSession обект няма да има никаква стойност под ключ “USER” и така основният сървлет ще разбере, че този потребител не е автентикиран:



Както видяхме проследяването на потребителската сесия става автоматично и в нея могат да се записват различни обекти под различни ключове, като се гарантира, че HttpSession обекта за всеки различен потребител е различен. Можем да задаваме времето на неактивност в милисекунди, за което една сесия изтича и се изтрива от сървъра чрез метода setMaxInactiveInterval() на класа HttpSession. Изтичането на сесиите (session expiration) е полезно от съображения за сигурност. След като един потребител е автентикиран веднъж, неговата автентикация важи само определено време, за да не може след като стане от компютъра някой друг да седне на негово място и да използва сесията му. Потребителският интерфейс на Web-приложенията обикновено освен автентикация и оторизация (login) предлага и изход от системата (logout), което прекратява потребителската сесия. Прекратяването на сесията на потребител в нашия пример може да стане като се изтрие стойността с ключ "USER" от сесията чрез session.removeAttribute("USER").

В предходната част от курса разгледахме как Java-сървлетите могат да извличат изпратените към тях параметри, изяснихме етапите от жизнения им цикъл и обяснихме как много потребители могат да работят едновременно и независимо един от друг с един сървлет като използват HTTP сесии. В тази последна част от курса ще изясним технологията Java Server Pages (JSP). Ще обясним основните тагове в JSP, ще изясним каква е връзката между JSP и сървлетите и ще покажем техники, които намаляват усилията за създаване на Web-приложения. Ще да дадем и цялостен пример за Web-приложение, което показва Java технологиите за Web-програмиране в действие.

Java Server Pages (JSP)

Java Server Pages (JSP) е технология, която позволява в статичен HTML документ да се вгражда програмен код на Java, който се изпълнява при заявка за достъп този документ. Фрагментите програмен код, вградени в HTML документа се ограждат със специални тагове и се наричат скриплети. HTML документи, съдържащи скриплети, се наричат JSP-страници или за по-кратко JSP-та. Когато Web-контейнерът изпълнява един JSP файл при клиентска заявка за достъп до него, се връща HTML документът, който се съдържа в този файл, като всички скриплети, вградени в него, се изпълняват се заместват с резултата получен от тях. Така един JSP документ, който е смесица между Java и HTML, след изпълнението си се превръща в динамично генериран чист HTML документ. Идеята е статичният HTML в един документ да си остане статичен, а програмен код на Java да се пише само за генериране на динамичните части от този документ, а не за целия документ. За разлика от сървлетите, в JSP-тата не е нужно статичният HTML текст да се отпечатва в изходния поток чрез извиквания от вида `out.println(…)`, защото вместо това може да се използва чист HTML текст. Това силно улеснява работата на разработчика, подобрява четимостта на кода и опростява поддръжката му.

JSP скриплети

Скриплетите в JSP-страниците се ограждат с таговете `<%` и `%>` съответно за начало и край. Ето един пример за JSP, което отпечатва текущата дата и час, използвайки скриплет:

```
<html>
    <head><title> Date JSP demo </title></head>
    <body>
        The date is:
        <% out.println(new java.util.Date()); %>
    </body>
</html>
```

Както се вижда от текста на това примерно JSP, то много прилича на обикновен HTML документ, само че съдържа скриплет, който се изпълнява динамично и отпечатва текущата дата и час чрез стандартните средства на Java.

Според стандарта за Java Server Pages във всички JSP-страници автоматично се създават следните обекти:

request – за достъп до заявката и параметрите изпратени чрез нея
response – за управление на отговора на заявката
out – изходен текстов поток за отговора на заявката
session – за управление на потребителските сесии
application – за достъп до данните, съхранявани във Web-приложението

За удобство на програмиста тези обекти са достъпни от всички скриплети в JSP-страницата. В нашия пример използвахме обекта out, чрез който отпечатахме текущата дата в изходния поток на JSP-страницата.

Технологията Java Server Pages предоставя на Web-разработчика освен скриплети и други тагове. Да разгледаме най-важните от тях.

JSP изрази

JSP изразите са съкратен начин за отпечатване на стойността на Java израз в изходния поток на отговора на сървлета. Това става чрез тага `<%= израз %>`, който е еквивалентен на скриплета `<% out.write(израз) %>`. За пример можем да дадем JSP-страница, която отпечатва числата от 1 до 100 и техните квадрати:

```
<html>
    <head><title> Squares JSP demo </title></head>
    <body>
        <% for (int i=1; i<=100; i++) { %>
            Square of <%= i %> is equal to <%= i*i %>.
            <br>
        <% } %>
    </body>
</html>
```

Забележете, че когато се използват конструкции за управление в Java като условни конструкции и конструкции за цикъл, които изискват тялото им да е в отделен блок, трябва винаги този блок да е ограден с отваряща и затваряща фигурна скоба, т.е. да започва с “`{`“ и да завършва с “`}`”. Дори ако се използва само 1 ред HTML за тяло на блок, е необходимо той да е ограден с фигурни скоби. Примерът по-горе демонстрира и още една възможност на JSP-тата – да се използва чист HTML в тялото на цикли и if-конструкции. Както се вижда, възможно е един цикъл да започва в един скриплет и да завършва в друг, а тялото му да е в чист HTML, разположен между двата скриплета. Това се обяснява с начина, по който JSP документите се трансформират в сървлети и по-точно в Java сорс-код на сървлети, който след това се компилира до класове.

Как JSP документите се преобразуват в сървлети

JSP-тата са файлове, които се трансформират в сървлети от Web-контейнера, който ги изпълнява и след това се изпълняват като най-обикновен сървлет. Поради тази причина всичко,

което знаем за сървлетите от предходните части на курса, важи и за JSP-тата. С малки изключения, целият JSP документ, включващ статичен HTML текст, скриплети и други JSP тагове, се трансформира в програмен код на Java и се записва в тялото на метод, който се извиква от метода `service(…)` на класа `HttpServlet`. Заради това JSP страниците отговарят както на GET така и на POST заявки по протокола HTTP. За простота можем да приемем, че Web-контейнерът, когато трансформира един JSP документ в сървлет, замества всички редове, представляващи чист HTML текст с програмен код, който отпечатва този текст в изходния поток на отговора на сървлета. Например първият ред от нашето JSP, съдържащ текста “`<html>`”, се заменя при трансформацията с програмен код подобен на `out.write ("<html>")`. Можем да си представим JSP-тата и по друг начин – като най-обикновени сървлети, в които има кратък вариант за отпечатване на чист HTML текст, без да се използва `out.println(…)` или `out.write(…)`. Разбира се, освен краткия начин за печатане на статичен HTML текст, JSP-тата ни спестяват и доста допълнителни усилия, щяха да са необходими, ако използвахме сървлети. С JSP не е нужно да се дефиниранира клас, който наследява `HttpServlet`, припокрива методите `doGet(…)`, `doPost(…)`, взема изходния поток от response-обекта и пише в него. Всички това става автоматично. Можем да считаме, че JSP е естествена крачка от развитието на сървлетите като технология, защото разширява техните възможности и същевременно значително намалява усилията за създаването на динамични HTML документи.

При първо извикване на едно JSP, Web-контейнерът го трансформира в Java сорс код на сървлет, компилира го и получава клас файл. След това зарежда този клас в паметта и го изпълнява. При всяко следващо извикване, ако JSP-то не е променено, то не се компилира повторно, а директно се изпълнява получения по първото извикване компилиран код. Ако JSP-то е променено, то автоматично се прекомпилирва. При JSP-тата не е необходимо да се растартира сървърът при всяка промяна, за да се види резултата от нея, както трябва да се прави при промяната на сървлет при повечето Web-контейнери. Понеже JSP-тата са сървлети, те имат същия жизнен цикъл, като сървлетите и могат да използват всичко, което знаем за тях, като например механизма за управление на потребителската сесия. При JSP-тата автоматично се създава обект с име session, който представя потребителската сесия и както знаем, е различен за всеки различен клиент на нашето Web-приложение. Можем да използваме този обект за съхранение на данни, свързани с текущия потребител и неговото взаимодействие с Web-приложението.

JSP декларации

JSP декларациите представляват фрагменти програмен код на Java, които се вмъкват в кода на генерирания от JSP-то сървлет, директно в класа на сървлета, който се получава. За разлика от скриплетите, които се вмъкват в тялото на метод, който се вика от метода `jspService(…)` на класа `HttpServlet`, JSP декларациите се вмъкват не в някакъв метод, а директно в тялото на класа. Синтактично JSP декларациите се отделят от статичния HTML текст чрез тага `<%! … %>`. Използват се най-често за дефиниране на методи, които след това могат да се извикват от скриплетите, а също и за деклариране на член-променливи в класа на сървлета, който се получава от даденото JSP. Ето един пример за JSP, което генерира и отпечатва 10 случайни числа, всяко от които е между 0 и 999:

```
<%!
private java.util.Random mRandomGenerator =
    new java.util.Random();

private int getRandomNumber(int range) {
    return mRandomGenerator.nextInt(range);
}
%>
<html>
    <head><title>Random numbers demo</title></head>
    <body>
        <% for (int i=1; i<=10; i++) { %>
            Random number #<%= i %> is
            <%= ""+getRandomNumber(1000) %>.<br>
        <% } %>
```

```
    </body>
</html>
```

Както се вижда от кода, в този пример с тага <!% … %> се декларира и инициализира обект от класа java.util.Random и метод в класа на сървлета, който връща случайно цяло число в зададен диапазон. След това този метод се използва от JSP израз, който отпечатва случайно число между 0 и 999. Може да се забележи, че използването на класа java.util.Random става чрез пълното име на класа, предшествано от името на пакета, в който стои този клас. При нормалното програмиране на Java в програмата могат да се включват пакети чрез ключовата дума import, следвана от име на пакет. След това могат да се използват класовете от включените пакети като се изписват само имената им без пакетите, на които те принадлежат. В JSP също има начин за import-ване на пакети. Това става с атрибутът <%@ page import="име_на_пакет" %>, който се слага обикновено в началото на JSP-страницата. Например следният ред в JSP документ:

```
<%@ page import="java.util.*" %>
```

е еквивалентен на реда

```
import java.util.*;
```

написан в началото на сървлета преди декларацията на класа, който се получава при трансформацията на JSP-то в сървлет. Чрез подобен атрибут на JSP документа може да се зададе и content-type-а на върнатия документ. Например ако искаме да върнем документ, който да се интерпретира от Web-браузъра на клиента като чист текст, а не като HTML, можем да напишем следното на един от началните редове на JSP документа:

```
<%@ page contentType="text/plain" %>
```

С подобни атрибути могат да се задават и други настройки на JSP-страницата. Например атрибутът

```
<%@ page session="false" %>
```

указва, че JSP страницата няма да използва сесия, с което се ускорява достъпът до нея и същевременно сървърът се натоварва по-малко. Тази настройка трябва да се слага във всички JSP страници, които не използват потребителската сесия (обекта session). Друг полезен атрибут на JSP страниците, който може да се задава по подобен начин, е страницата за обработка на грешки (error page). Ако в началото на една JSP страница се сложи ред, който съдържа

```
<%@ page errorPage="някое_релативно_URL" %>
```

то всяко изключение (exception), възникнало по време на изпълнение на JSP-то, което не е обработено от това JSP, се предава на зададената страница за обработка на грешки. Задачата на тази страница за обработка на грешки е да покаже грешката във формат, разбираем за потребителя и евентуално да се погрижи да уведоми администратора за възникналия проблем. Всяка error страница трябва да съдържа тага <%@ page isErrorPage="true" %>, който указва, че това е страница за обработка на грешки. В такива страници е достъпен още един допълнителен обект exception, който съдържа последното възникнало изключение, което описва грешката.

JSP и JavaBeans

Според JavaBeans спецификацията bean-овете представляват обикновени Java класове, които отговарят на следните допълнителни условия: имат конструктор без параметри; нямат публични член-променливи; могат да имат свойства (properties), които са достъпни чрез публични методи с имена getXXX() и setXXX(…), където XXX е името на съответното property. В JSP страниците могат да се създават и използват Java bean-ове чрез тага <jsp:useBean … />. Например тага:

```
<jsp:useBean id="userInfo" class="com.nakov.example.UserInfo" />
```

декларира екземпляр на Java bean с име userInfo от класа com.nakov.example.UserInfo, който е достъпен от всички скриплети на JSP-то. Тази декларация е почти еквивалентна на обикновеното инстанциране на клас, което в скриплет може да стане чрез следния код:

```
<% com.nakov.example.UserInfo userInfo = new com.nakov.example.UserInfo(); %>
```

За разлика от директното инстанциране на класове, тагът <jsp:useBean … /> дава доста допълнителни възможности. Една от тези възможности е задаването на обхват на действие за bean-овете чрез атрибута “scope”. Този обхват може да бъде текущата страница (page scope), текущата заявка (request scope), текущата потребителска сесия (session scope) или цялото Web-приложение (application scope). Един bean се създава винаги при първото му използване, а след това не се унищожава, докато не излезе от обхвата, с който е дефиниран. Например ако се използва bean с

обхват текущото приложение, той ще се създаде при първото му използване и ще е достъпен от всички JSP-та и сървлети в приложението. Класът на bean-а ще се инстанцира само веднъж в рамките на приложението и ще се унищожи при спиране на това Web-приложение или при спиране на сървъра. Ако се използва bean с обхват текущата сесия, той ще се създава при всяко първо извикване в рамките на всяка нова сесия и ще се унищожава при унищожаване на сесията, т.е. този bean ще има по една инстанция за всяка потребителска сесия на Web-приложението. Обхватът request и обхватът page много си приличат по това че са краткотрайни – важат само в рамките на едно извикване. Bean-овете с обхват page съществуват само през времето, в което се изпълнява JSP-страницата и се унищожават при приключване на нейното изпълнение. Bean-овете с обхват request съществуват през цялото време на подготвянето на отговора на клиентската HTTP заявка, дори ако този отговор се генерира в резултат от последователното изпълнение на няколко JSP-та.

За достъп до полетата на един bean (неговите properties), има два JSP тага: <jsp:getProperty … /> и <jsp:setProperty … />. Те са еквивалентни на директния достъп до полетата на bean-а. Например изразът

```
<jsp:getProperty name="userInfo" property="name" />
```

е еквивалентен на израза

```
<%= userInfo.getName() %>
```

И двата израза отпечатват името на потребителя, който се описва от bean-а userInfo. Ползата от таговете <jsp:getProperty …/> и <jsp:setProperty … /> е това, че са в XML формат, което ги прави по-лесни за използване от човек, който не е програмист. Освен това те имат и допълнителни възможности. Ето и пример за задаване на стойност на поле в bean:

```
<jsp:setProperty name="userInfo" property="password"
value='<%= request.getParameter("userPassword") %>' />
```

Забележете, че в стойността на атрибута value на тага <jsp:setProperty … /> може да се използват JSP изрази, а не само константен текст.

Едно от полезните неща, от които може да се възползва програмистът, който използва JavaBeans съвместно с JSP при разработваното Web-приложение, е зареждането на полетата на bean-ове от параметри, изпратени към дадена JSP страница. Това може да стане чрез атрибута "param" на тага <jsp:setProperty … />. Например следният код:

```
<jsp:setProperty name="userInfo" property="password" param="userPassword" />
```

зарежда в полето password на bean-а userInfo стойността, записана в параметъра с име userPassword на заявката към страницата. Ако типът на полето в bean-а е числов, се прави автоматично конвертиране в число на текстовата стойност, съдържаща се в параметъра.

Друго предимство при използването на JavaBeans съвместно с JSP е, че може да се зададе автоматично зареждане на всички полета на даден bean от параметри на заявката със същите имена. Например ако имаме bean-а userInfo, който съдържа полетата name, password и age, можем да ги заредим от изпратените към страницата параметри чрез следния код:

```
<jsp:setProperty name="userInfo" property="*" />
```

За да е успешно зареждането, е необходимо към страницата да са изпратени параметри с имена name, password и age. Ако имената на полетата и имената на изпратените параметри не съвпаднат, при зареждането някои полета ще останат без стойност. При много на брой параметри и полета на bean-а, в който тя трябва да се запишат, този таг е много полезен, защото спестява голямо количество код и механичния труд, необходим за написването на този код.

От всичко, изтъкнато до момента, можем да си направим извода, че използването на Java bean-ове е много полезно за отделяне на логиката от визуализацията в Web-приложенията. Чрез съвместното използване на JavaBeans и JSP-та се дава възможност на Web-дизайнерът да създаде перфектния дизайн за нашето Web-приложение, без да знае Java и без да познава детайлите на JSP програмирането, а на програмиста се дава възможност да пише части от кода в отделни класове (bean-ове) извън JSP-то, като така концентрира вниманието си върху тях, а не върху HTML таговете.

Друг полезен таг в JSP страниците е тагът <%@ include file="relative_url" %>. Той позволява включването на съдържанието файл на текущата позиция в дадена JSP страница. Включването става по време на трансформирането на JSP страницата в сървлет. Този таг е особено подходящ когато Web-приложението съдържа много JSP страници, съдържащи общи фрагменти. Например ако трябва в началото на всяка страница от нашето Web-приложение да има меню, бихме могли да

отделим кода, който създава това меню в отделен файл и да го включваме във всеки JSP файл с тага `<%@ include … %>`. Тази възможност позволява повторното използване на вече написани фрагменти код (code reuse), което при големи проекти е много често използвана техника. Например може в началото на JSP документа да се включи следния ред:

```
<%@ include file="menu.jsp" %>
```

Той включва съдържанието на файла menu.jsp в текущата JSP страница по време на компилация. Освен този таг, за включване на фрагмент код в текущата JSP страница има и още един подобен таг: `<jsp:include page="relative_url" />`. При включване чрез `<%@ include … %>` включеният файл се прочита веднъж при първото изпълнение на JSP-то и след това дори да бъде променен, промените не се отразяват на JSP-то (static include). При включване на файл чрез `<jsp:include … />` включеният файл се изпълнява при всяка заявка към JSP страницата и резултатът от него се вмъква в страницата (dynamic include). Така, ако включеният файл бъде променен, промяната се отразява и на всички JSP-та, които го включват. Освен това чрез `<jsp:include … />` могат да се включват сървлети, CGI скриптове и други ресурси, достъпни чрез зададеното URL, а не само фрагменти от JSP документи. Ето и пример за включване на заглавен фрагмент в началото на JSP страница:

```
<jsp:include page="header.jsp" flush="true"/>
```

Атрибутът `flush="true"` е задължителен и трябва винаги да се включва при използване на `<jsp:include … />` тага. Стойност false не е допустима.

Още един полезен таг в JSP стандарта е тагът за пренасочване към друго JSP `<jsp:forward page="relative_URL"/>`. При изпълнение на този таг, като резултат от заявката на клиента се връща резултатът от изпълнението на посоченото URL. Има голяма разлика между пренасочване чрез `request.sendRedirect(…)` (browser redirection) и `<jsp:forward … />` (server redirection). Методът `request.sendRedirect(…)` просто казва на браузъра да зареди посоченото URL вместо това URL, което е поискал. Това става като сървърът върне отговор с код 302 на HTTP заявката (document temporary moved). Такова пренасочване е еквивалентно на това потребителят да напише посоченото URL в address bar-а на браузъра и да го зареди. Пренасочването с `<jsp:forward … />` работи по друг начин. При него браузърът не разбира, че на сървъра се е извършило пренасочване, а просто получава резултата от изпълнението на URL-то, към което е направено пренасочване с `<jsp:forward … />`. В такъв случай в address bar-а на браузъра URL-то не се променя.

Цялостен пример за Web-приложение

За да илюстрираме всички техники, с които се запознахме, ще напишем едно цялостно многопотребителско Web-приложение и ще го изпълним с Web-контейнера Tomcat. Да си поставим за задача разработката на много прост дискусионен форум. Приложението трябва да има две страници – едната за влизане във форума, а другата за четене на съобщенията и добавяне на нови съобщения. За влизане във форума се изисква потребителско име и парола. Ако въведеното име съвпада с въведената парола, системата трябва да пуска потребителя на страницата за четене и добавяне на съобщения. В тази страница трябва да се извеждат в таблица всички съобщения и да има форма за добавяне на ново съобщение. Съобщенията се състоят от тема и съдържание. За леснота съобщенията могат да се пазят само в паметта на приложението, т.е. се губят при рестартиране на сървъра. Системата не трябва да позволява достъп до форума на неоторизирани лица, който не са влезли през началната страница. Ето как изглежда едно възможно решение на задачата:

```
<%-- File name: login.jsp --%>
<%@ page contentType="text/html; charset=windows-1251" %>
<html>
<head><title>Login</title></head>
<body>
    <%@ include file="header.jsp" %>
    <div align="center">
    <%
    String userName =
        request.getParameter("user");
    String password =
        request.getParameter("pass");
```

```
    if ((userName!=null) && (password!=null)
            && (userName.length()>0)
            && (userName.equals(password))) {
        session.setAttribute("USER", userName);
        response.sendRedirect("main.jsp");
    }
    if (userName!=null) {
    %>
        Невалиден логин. Опитайте отново.<br>
    <%
    }
    %>
    <form action="login.jsp">
        <input type="text" name="user"><br>
        <input type="password" name="pass"><br>
        <input type="submit" value="Влез">
    </form>
    </div>
</body>
</html>
```

```
<%-- File name: main.jsp --%>
<%@ page contentType="text/html; charset=windows-1251" %>
<%@ page import="java.util.*" %>
<%!
    private String getMsgsHtml(ArrayList msgs)
    {
        if (msgs.size() == 0) {
            return "Няма съобщения.";
        }
        String resultHtml =
            "<table border=1 width=100% " +
            "cellspacing=0><tr><td>Тема</td>" +
            "<td>Съобщение</td></tr>\n";
        for (int i=0; i<msgs.size(); i++) {
            Message msg = (Message) msgs.get(i);
            resultHtml = resultHtml +
                "<tr><td>" + Utils.htmlEscape(
                msg.getSubject()) +
                "</td><td>" + Utils.htmlEscape(
                msg.getContents()) +
                "</td></tr>\n";
        }
        resultHtml = resultHtml + "</table>\n";
        return resultHtml;
    }
%>
<html>
<head><title>View Forum</title></head>
<body>
    <%@ include file="header.jsp" %>
    <div align="center">
    <%
    String userName = (String)
        session.getAttribute("USER");
    if (userName==null) {
    %>
        Нямате достъп до тази страница.<br>
        Моля <a href="login.jsp">влезте</a>
        в системата.
    <%
    } else {
        ArrayList msgList = (ArrayList)
            application.getAttribute("MESSAGES");
        if (msgList == null)
            msgList = new ArrayList();
        %>

        <%-- Add the new message if any --%>
        <jsp:useBean id="msg" class="Message" />
        <jsp:setProperty name="msg" property="*" />
        <%
            if (msg.getContents() != null) {
```

```
                msgList.add(msg);
                application.setAttribute(
                    "MESSAGES", msgList);
            }
        %>

        <br>
        <%-- Print all the messages --%>
        <%= getMsgsHtml(msgList) %>
        <br>

        <%-- Add new message form --%>
        <form action="main.jsp">
        <table border="0">
        <tr><td>Тема:</td><td>
        <input type="text" name="subject">
        </td></tr>
        <tr><td>Съобщение:</td><td>
        <input type="text" name="contents">
        </td></tr>
        <tr><td> </td><td>
        <input type="submit" value="Добави">
        </td></tr>
        </table>
        </form>
    <%
    }
    %>
    </div>
</body>
</html>
```

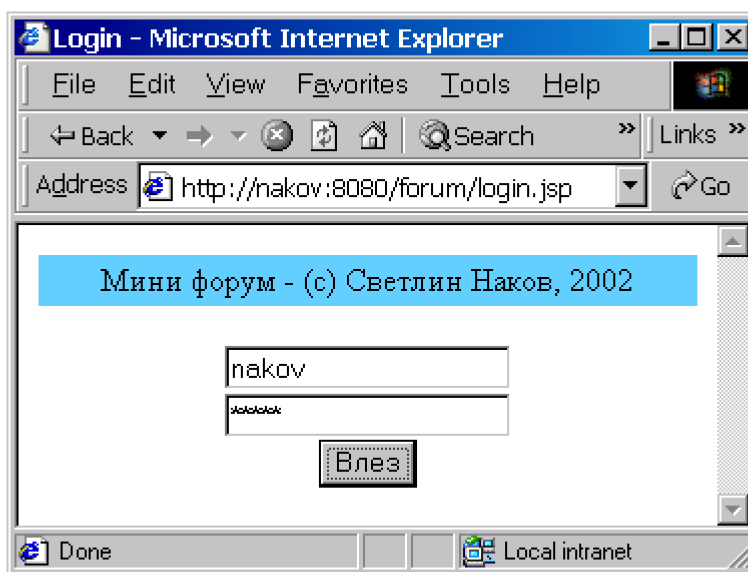
```
<%-- File name: header.jsp --%>
<table border="0" bgcolor="#66CCFF" width="100%">
<tr><td align="center">
Мини форум - (c) Светлин Наков, 2002
<%
    String currentUser = (String)
        session.getAttribute("USER");
    if (currentUser != null)
        out.write(" - потребител: " +
        Utils.htmlEscape(currentUser));
%>
</td></tr>
</table>
```

```
// File name: Message.java
public class Message {
    private String mSubject;
    private String mContents;

    public String getSubject() {
        return mSubject;
    }
    public void setSubject(String subject) {
        mSubject = subject;
    }
    public String getContents() {
        return mContents;
    }
    public void setContents(String contents) {
        mContents = contents;
    }
}
```

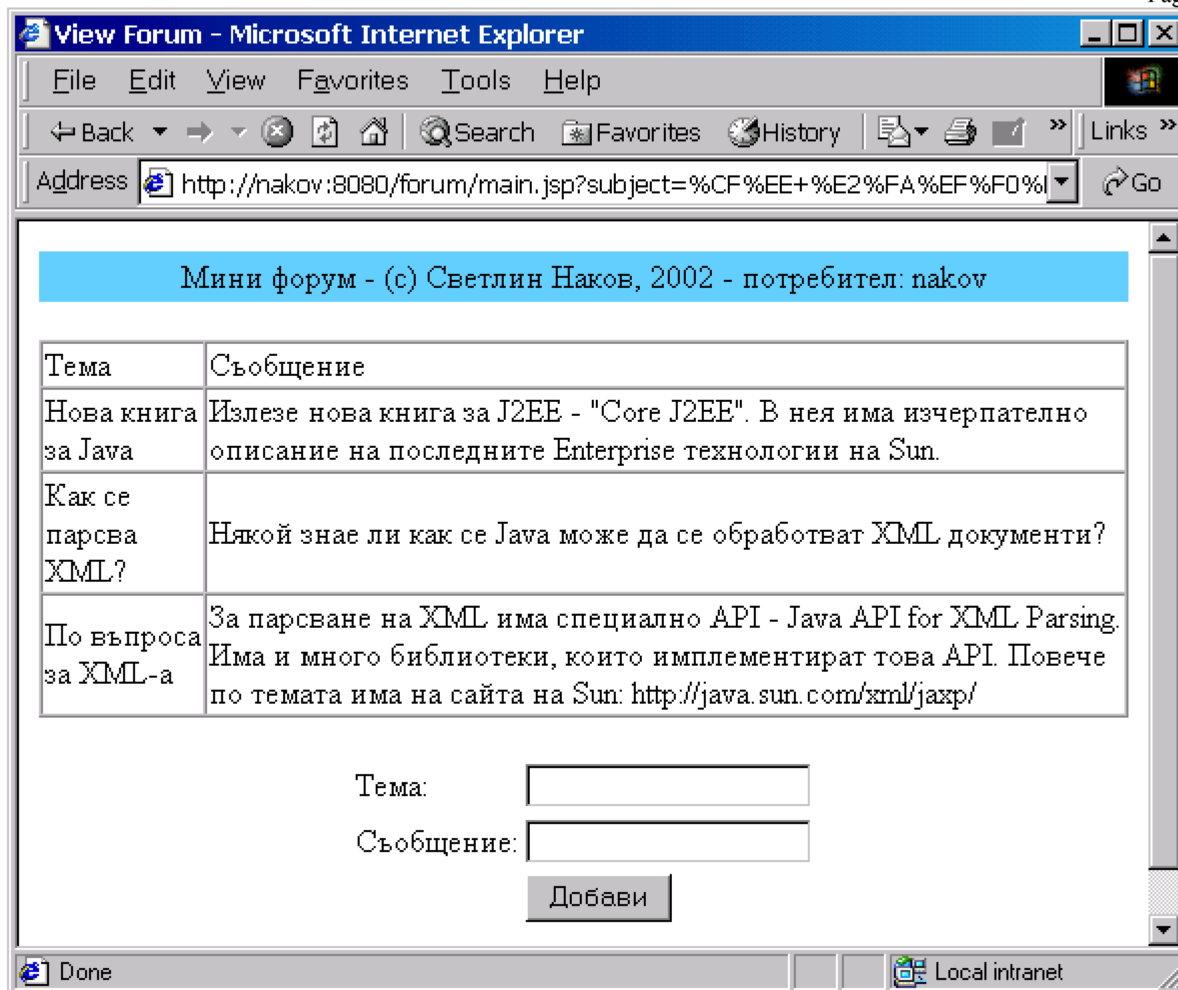
```
// File name: Utils.java
public class Utils {
    public static String htmlEscape(String s) {
        StringBuffer out = new StringBuffer();
        for (int i=0; i<s.length(); i++) {
            char ch = s.charAt(i);
            if (ch == '\'')
                out.append("&#39;");
```

```
            else if (ch == '\"')
                out.append("&#34;");
            else if (ch == '<')
                out.append("&lt;");
            else if (ch == '>')
                out.append("&gt;");
            else if (ch == '&')
                out.append("&amp;");
            else
                out.append(ch);
        }
        return out.toString();
    }
}
```

За да се накара приложението да заработи под Tomcat, трябва да се направи следното: Да се създаде поддиректория с име `forum` в `%TOMCAT_HOME%/webapps` и в нея да се запишат всички JSP файлове. В същата тази директория `forum` трябва да се създаде поддиректория с име `WEB-INF`, а в нея поддиректория `classes`. Забележете, че има значение между малки и главни букви, особено ако искаме нашето Web-приложение да работи и под Linux. В поддиректорията `classes` трябва да се копират всички Java файлове и да се компилират до `class` файлове. Това е всичко, което е необходимо за стартиране на форума. Структурата на директориите не е подбрана случайно, а се описва от спецификацията за Web-приложения на стандарта за J2EE. Таи спецификация гарантира, че ако директорията `forum` се копира заедно със всичките й поддиректории на друг Web-контейнер, приложението ще заработи по същия начин. След като се стартира сървърът Tomcat, приложението е достъпно от адрес `http://localhost:8080/forum/login.jsp`. Ето и изглед от началния екран (`login.jsp`):



След успешно влизане във форума, се преминава на екрана за разглеждане и добавяне на съобщения (`main.jsp`):



Да разгледаме по-подробно кода на нашето примерно Web-приложение. Началната страница `login.jsp` започва със задаване на content-type-а и набора от символи, които да се използват при показване на страницата. Избрана е стандартната кодова таблица windows-1251, която е единствената, използвана в България за работа с кирилица под Windows. Тази кодова таблица се поддържа от всички стандартни Web-браузъри и затова кирилицата в страницата би трябвало да се покаже правилно без да е нужно потребителят да задава никакви допълнителни настройки. Следва стандартно начало на HTML документ, след което статично се включва файлът `header.jsp`. Този файл се включва в началото на всяка страница на приложението и служи за изобразяване на стандартна заглавна част, в която пише името на приложението, името на текущия потребител и още някаква друга допълнителна информация. Следва извличане на параметрите потребителско име и парола, ако са изпратени такива. Понеже `login.jsp` работи в два варианта – без параметри и с параметри, се разглеждат съответно няколко случая. Ако са изпратени потребителско име и парола, те се валидират (за простота просто се проверява дали съвпадат и не са празни низове) и ако валидацията е успешна, в сесията се записва името на валидирания потребител под ключ "`USER`" и браузърът на клиента се препраща към основната страница на приложението (`main.jsp`). Ако валидацията е неуспешна, се отпечатва съобщение за грешка и се извежда формата за въвеждане на потребител и парола. Ако `main.jsp` се извика без параметри, се извежда същата форма за въвеждане на потребител и парола, но без никакво съобщение. Тази форма е настроена да изпраща данните, въведени от потребителя, към същата страница, в която се намира самата тя (`index.jsp`). Така потребителят може да прави последователно много опити за влизане в системата, но достига до главната страница на форума само при успешна валидация. Ако той се направи на хитър и въведе директно URL-то на основната страница `http://localhost:8080/forum/main.jsp`, системата няма да го пусне, защото няма да намери име на потребител, записано в сесията под ключа "`USER`".

Нека сега разгледаме същината на приложението – главната страница `main.jsp`. Тя също може да се вика с параметри или без параметри. Ако се извика без параметри, тя извежда всички съобщения от списъка. Ако се извика с параметри тема и съобщение, тя добавя това съобщение в

списъка със съобщенията и извежда този списък. Подобно на login.jsp първоначално се задава кодовата таблица, което осигурява правилното извеждане на кирилицата. След това се показва заглавната част на страницата чрез статичното включване на файла header.jsp. Следва вземане на името на валидирания потребител от текущата сесия. Това се прави като се извлича стойността записана под ключ с име "USER" в сесията. Ако потребителят е преминал успешно през login.jsp, би трябвало такава стойност да има. Ако потребителят не е валидиран, такава стойност няма да има, т.е. ще се получи стойност null. В този случай потребителят се уведомява че няма право на достъп и се подканва да влезе в системата през началната страница. Ако потребителят е валидиран, има два случая – или към страницата е изпратена заявка за добавяне на ново съобщение, или страницата е поискана без параметри. За извличане на параметрите се използва JavaBean класа Message, който се инстанцира с тага <jsp:UseBean … />. След това параметрите се записват в bean-а чрез техниката за автоматично прехвърляне на параметри с тага <jsp:setProperty … />. Забележете, че bean-а Message има полета, които точно съответстват на имената на очакваните параметри. Ако е имало изпратен параметър за съдържание на съобщение, в полето contents на bean-а ще има стойност, различна от null. Ако стойността е null, се счита, че страницата е извикана с цел показване на всички съобщения, а не с цел добавяне на ново съобщение. За съхранение на съобщенията в приложението се използва списък (java.util.ArrayList), който се съхранява в приложението под ключ "MESSAGES". Забележете, че списъкът със съобщенията се съхранява в приложението, което го прави достъпен за всички потребители, а не се съхранява в сесията, както името на активния потребител. Възможно е в приложението да няма ключ под име "MESSAGES". Това означава, че все още никой потребител не е добавял никакви съобщения. Ако списък със съобщения няма, той се създава и първоначално е празен. След това ако е изпратено като параметър ново съобщение, то се добавя в списъка и стойността под име "MESSAGES" в приложението се актуализира. Независимо дали е добавено ново съобщение или не, всички съобщения се извеждат. За целта се използва процедура, дефинирана чрез JSP декларация с тага <%! … %>. Тя поема като параметър списък от съобщения и връща като резултат HTML текст, който представлява визуализация на тези съобщения във вид на таблица. След кода за визуализацията на съобщенията в JSP страницата е разположена HTML формата за добавяне на нови съобщения. Тази форма съдържа само две полета и бутон за добавяне на съобщение, който изпраща данните, попълнени от потребителя към същата JSP страница (main.jsp). Както вече обяснихме, тези параметри се записват в bean и се обработват по подходящ начин след това. Остана да обясним как се визуализират съобщенията във вид на HTML. Методът getMsgsHtml(…), който се дефиниран като частен за класа на JSP страницата main.jsp, първо проверява дали списъкът, който му е подаден като параметър не е празен. Ако е празен, връща подходящо съобщение, а в противен случай създава таблица и извежда в нея елементите на списъка със съобщенията използвайки for-цикъл. Едно много важно нещо, което се прави при отпечатването на съобщенията в таблицата, която се генерира, е да се заменят непозволените за HTML документите символи с техния еквивалент в HTML (HTML escaping). Става въпрос за символи като <, >, & и някои други, които могат да повредят HTML документа, ако се извеждат в него без да се escape-ват. Например ако някой добави във форума съобщение със съдържание "</td></tr></table>грешка!", може да повреди таблицата и да предизвика писане на текст извън нея. В HTML има специален начин за избягване на опасните символи, наречен HTML escaping, който трябва да се използва винаги, когато се извеждат низове, които биха могли да съдържат опасни символи. Въпреки, че този проблем възниква почти в 100% от Web-приложенията, разработвани с Java, в JSP/Servlet API-то няма стандартен метод, който извършва HTML escaping. Колкото и да е странно, стандартен еквивалент на PHP функцията html_escape(…) няма и ние трябва сами да си го напишем. Съществува един стандартен клас java.net.URLEncoder, но той служи за друг вид escaping – URL escaping, който се използва за кодиране на информацията при изпращане на HTML форми. За да може да се използва от всички JSP-та на приложението, този метод е дефиниран като статичен в отделен клас с име Utils. Ако някога ви се налага HTML escaping, което е доста вероятно, ако пишете Web-приложение, използвайте наготово метода htmlEscape(…), който е даден в края на сорс-листинга, за да не губите време да си го пишете сами или да го търсите в Интернет.

По листинга по-горе следва реализацията на файла `header.jsp`, съдържащ фрагмент от код, който е предназначен да стои в началото на всяка страница от приложението. В този код се проверява дали в сесията има верифициран потребител и ако има, освен стандартният заглавен текст, се отпечатва и името на активния потребител.

Примерното приложение има много ограничена функционалност, защото изискванията към него са да бъде доста кратко и същевременно да илюстрира възможно по-голяма част от JSP таговете и техниките описани в настоящата статия. Оставяме на читателя да се опита да го подобри и разшири, като вярваме, че със знанията придобити от настоящата поредица статии "Интернет програмиране с Java", това няма да го затрудни.

В заключение трябва да отбележим, че JSP технологията за разработка на Web-приложения е една от водещите в световен мащаб и е популярна не по-малко от ASP, PHP, Perl и ColdFusion. Наблюдава се тенденция за по-големите и по-сложните приложения да използва именно JSP вместо Perl и PHP, защото J2EE платформата отговаря на всички нужди, които имат тези приложения – надеждност, сигурност, скалируемост, разпределеност, разширяемост и т.н.

***Внимание:* Учебникът е в работен вариант! Възможно е наличието на грешки.**

14.02.2003, София

Светлин Наков