Светлин Наков

ИНТЕРНЕТ ПРОГРАМИРАНЕ с JAVA

Faber

Интернет програмиране с Java

Светлин Наков

Софийски университет „Св. Климент Охридски”
Българска асоциация на разработчиците на софтуер

София, 2004

Интернет програмиране с Java

Web-site: www.nakov.com
E-mail: inetjava-book@nakov.com

Официален сайт:
www.nakov.com/books/inetjava/

ISBN 954-775-305-3

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» ***Светлин Наков*** е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» ***Мартин Кулов*** е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

www.devbg.org

Българската асоциация на разработчиците на софтуер е нестопанска организация, която подпомага професионалното развитие на българските софтуерни разработчици чрез различни дейности, насърчаващи обмяната на опит между тях и усъвършенстването на техните знания и умения в областта на проектирането и разработването на софтуер.

Асоциацията организира конференции, семинари, симпозиуми, работни срещи и курсове за обучение по разработка на софтуер и софтуерни технологии.

Отзив за книгата „Интернет програмиране с Java”

Светлин е опитен програмист, многократно доказан специалист по съвременни софтуерни технологии, талантлив лектор. Неговите курсове в Софийски Университет са сред най-високо ценените, най-полезните и най-посещаваните. Със своите лекции и презентации по научни и технически конференции и семинари той неведнъж е събирал аплодисментите на начинаещи и професионални софтуерни разработчици.

В духа на своята творческа натура и нестихващ академичен идеализъм през 2002 Светлин, заедно със свои колеги, разработи курса „Интернет програмиране с Java”. По време на обучението на студентите от Факултета по математика и информатика на Софийски университет, които избраха да слушат този курс, Светлин и колегите му създадоха лекции по “Интернет програмиране с Java”, които по-късно станаха основа на серия от статии в списание “PC Magazine/Bulgaria”. В последствие, с много труд и усилия тези учебни материали достигнаха един завършен академичен вид и станаха основа на настоящия учебник.

Книгата „Интернет програмиране с Java” е едно отлично въведение в най-важните аспекти на програмирането с Java за Интернет. В нея се обръща внимание на проблемите на многонишковото програмиране и синхронизацията, разработката на Java приложения, които комуникират по протоколите TCP/IP, създаването на Java аплети, комуникацията между аплет и сървър, разработката на Web-приложения с технологиите Java Servlets и Java Server Pages (JSP) и изпълнението им на сървъра за Web-приложения Tomcat.

Настоящата книга е официален учебник за дисциплината „Интернет програмиране с Java”, преподавана от Светлин Наков и ръководения от него екип във факултета по математика и информатика на Софийски Университет „Св. Климент Охридски” през 2004 г.

доц. д-р Магдалина Тодорова
Софийски Университет „Св. Климент Охридски”

Съдържание

Предговор

Ако по принцип не четете предговорите на книгите, пропуснете и този. И той е така скучен, както всички други. В него ще бъдат изяснени стандартните въпроси „за кого е тази книга", „какво включва тя", „какво се очаква от читателите", „какво се очаква да научим от книгата" и разни други неща все в тоя дух.

Няма да ви занимаваме с празни встъпителни приказки в стил „колко много Интернет е навлязъл в живота ни", „кога е възникнал Интернет", „колко много има нужда от Интернет приложения и Интернет програмисти", „кой е създал Java", "колко е велика Джавата", „колко е всемогъщ Интернета", „сами ли сме във вселената" и „какъв е смисъла на живота". Вместо това направо ще пристъпим към същината.

За кого е тази книга

Настоящият учебник по "Интернет програмиране с Java" е предназначен за всички, които се интересуват от разработка на Интернет-ориентирани приложения и програмиране на Java.

Какво съдържа тази книга

В настоящата книга се обръща внимание на най-важните технологии от областта на Интернет-ориентираното програмиране с езика и платформата Java:

- програмиране със сокети – разработка на Java приложения, които комуникират по Интернет и Интранет мрежи чрез протоколите TCP/IP;

- Java аплети – разработка на малки Java приложения с графичен потребителски интерфейс, които се вграждат във Web страници и се изпълняват от Web-браузърите на потребителите;

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

Какво се очаква от читателя

За да бъде разбран материалът, е необходимо читателите да имат основни познания по обектно-ориентирано програмиране, да са запознати с езика Java, да имат обща представа за организацията на Интернет и начални знания по HTML. Не е необходимо добро владеене на езика Java. Тази книга учи на концепции, базови знания и технологии, а не на Java.

Какво да очакваме да научим от тази книга

В никакъв случай тази книга не може да ни направи експерт по Интернет програмиране с Java. Целта на книгата е да ни даде начални, базови знания, които да ни послужат като основа в развитието ни като програмисти, софтуерни специалисти и Интернет разработчици.

Известно е, че когато един програмист има начални знания по някоя технология, той много бързо и лесно може да ги доразвие и да достигне високо професионално ниво. Много по-трудно е, обаче, ако започне от нулата. Това е и смисъла да учим много неща, които не използваме директно в работата си. Има смисъл да познаваме много технологии, за да избираме правилната, когато имаме конкретен проблем за решаване, но няма смисъл да сме експерти в нещо, което не използваме.

Като че ли най-добрата стратегия за един добър софтуерен разработчик е да разбира по-малко от много и най-разнообразни технологии, но да е добър специалист в тези от тях, които използва. В момента, в който един разработчик спре да учи нови неща и се пусне по течението със самочувствието, че вече знае прекалено много и няма нужда от повече, той вече не е добър разработчик.

Тази книга няма да ви направи специалисти нито по Интернет технологии, нито по Java, нито по Web-програмиране, но ще ви даде една безценна основа, с която ще можете да се развивате в тази посока, независимо от езиците за програмиране и платформите, с които ще работите.

Как са представени темите

Темите са разделени в три глави – сокет програмиране, Java аплети и Web-приложения. Всяка от тях започва с въведителна част, в която се изясняват основните концепции за съответната технология, а след това малко по малко се навлиза в материята, структурирано, последователно и с **много примери**.

Обичате примерите, нали? Какво е една книга за програмиране без примери? Въобще някой чете ли текста, когато търси нещо и това нещо го има в примерите?

Стремежът ни да разгледаме възможно повече основополагащи знания ни кара да започнем с изясняване на основните идеи и технологии, върху които е изградена световната мрежа Интернет.

Поради очакването, че не всички читатели познават добре Java, в началото обръщаме внимание и на някои базови знания от Java платформата като средства за вход/изход и средства за многонишково

програмиране и синхронизация. По-нататък даваме решение на един класически синхронизационен проблем – проблемът „производител-потребител", на който ще се натъкваме след това много пъти.

След въведителните теми пристъпваме към програмирането със сокети. Ще разгледаме клиент/сървър комуникацията по протокол TCP и ще разгледаме много примери, като малко по-малко ще увеличаваме сложността им, докато достигнем до проблема за създаване на многопотребителски чат сървър, при който нещата не са съвсем прости. По-нататък ще продължим с протокола UDP, ще се запознаем с multicast-сокетите и ще завършим главата с темата за достъп до ресурси по URL.

Във втора глава ще се занимаваме с технологията на Java аплетите. Ще навлезем в тайните на библиотеката AWT, нейният програмен модел и ще видим как да я използваме при създаването на аплети. Ще си изясним особеностите на аплетите, тяхната среда за изпълнение, жизненият им цикъл и ще завършим с проблемите на сигурността и начините за комуникация между аплет и Web-сървър.

В следващата глава ще навлезем в Web-програмирането. Първоначално ще изясним неговите основни концепции, базовите понятия, свързани с него, протоколите, програмния модел, езиците за описание на Web-съдържание и технологиите за динамично генериране на Web-съдържание. След това ще представим технологията на Java сървлетите, ще изясним как се използва сървъра за Web-приложения Tomcat и как да изпълняваме сървлети и Web-приложения с него. След това ще пристъпим към техниките за извличане на параметри, подадени от клиента и средствата за управление на потребителски сесии. Ще изясним и технологията Java Server Pages и таговете, свързани с нея. Накрая ще изясним цялостната концепция за Web-приложения на платформата J2EE и ще дадем пример за едно такова приложение.

Сайтът на книгата

Официалният сайт на книгата е на адрес:

http://www.nakov.com/books/inetjava/

В него можете да намерите допълнителни материали, ресурси и информация, свързана с учебния материал в книгата, сорс-кода от примерите, информация за самата книга, форма за изпращане на грешки и дискусионен форум.

Моля отправяйте всички коментари, въпроси и предложения във форума на книгата, а не ми ги изпращайте по e-mail. Всеки ден

получавам стотици писма и има вероятност да не успея да ви отговоря, ако отправите технически въпрос директно по e-mail.

Как се роди тази книгата

Настоящата книга е резултат от дългогодишната работа на автора по съставянето на лекции по „Интернет програмиране с Java" за едноименния курс, който се провежда от 2002 г. в Софийски университет „Св. Климент Охридски". Книгата успя да събере в себе си най-важното от целия опит на преподавателския колектив в областта на Интернет програмирането и да го синтезира в една кратка и достъпна за българските студенти форма.

Благодарности

Авторът изказва най-сърдечните си благодарности на всички негови колеги и приятели, които го подкрепяха и му помагаха по време курсовете „Интернет програмиране с Java" във Факултета по Математика и Информатика на Софийски Университет „Св. Климент Охридски" и които го насърчаваха през цялото време на работата му върху книгата:

Борис Червенков

Николай Недялков

Красимир Семерджиев

Димитър Георгиев

Лъчезар Цеков

Райчо Минев

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» ***Светлин Наков*** е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» ***Мартин Кулов*** е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Глава 1. Разработка на Java приложения, които си комуникират по TCP/IP мрежи

В настоящата глава ще разгледаме средствата на Java за разработка на Интернет приложения, които си комуникират със сокети по стандартните за Интернет протоколи TCP/IP.

В началото ще направим кратък преглед на TCP/IP мрежите с който ще въведем базови понятия като IP адрес, сокет, порт, клиент, сървър, протокол, услуга. Ще разгледаме двата основни транспортни протокола TCP и UDP. Ще разгледаме най-общо как работи световната мрежа Интернет и какви услуги предоставя тя.

За читателите, които не са добре запознати с Java платформата, ще направим кратък преглед на средствата на Java за осъществяване на входно-изходни операции, които ще ни трябват след това при разработката на клиентски и сървърски приложения, които си комуникират по сокети. Ще обърнем внимание на двата основни типа потоци в Java – текстови и бинарни.

Ще разгледаме в дълбочина средствата на Java за създаване на многонишкови (multithreading) приложения и най-вече проблемите със синхронизацията на достъпа до общи ресурси. Ще дадем решение на класическия проблем „производител – потребител" на базата на синхронизационните обекти „монитори".

След всички въведения ще имаме вече достатъчно познания, за да продължим по-нататък с темата за изграждане на приложения, които си комуникират по надеждни двупосочни поточно-ориентирани канали изградени на базата на TCP сокети.

Ще обърнем внимание и на средствата на Java за ненадеждно изпращане на единични пакети с информация между две приложения по протокол UDP, след което ще видим как можем да използваме механизмите на multicast инфраструктурата за да реализираме комуникация базирана на абонамент и групова доставка на съобщения.

В края на главата ще се запознаем със средствата на Java за извличане на ресурси от глобалната разпределена информационната система WWW (World Wide Web). Ще обясним какво е URL и как ще демонстрираме леснотата, с която в Java можем да извличаме ресурс от WWW по неговия URL адрес.

1.1. Как работи Интернет. Основи на TCP/IP мрежите

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 от организацията на TCP/IP мрежите, които имат пряко отношение към мрежовото програмиране с Java. Предполагаме, че читателят има поне начални познания по организация на Интернет и затова няма да обясняваме подробно някои общоизвестни термини като например „сървър", "протокол" и „мрежа".

7-слоен OSI модел на компютърните мрежи

Според световно възприетите стандарти за компютърни мрежи на организацията IEEE (Institute of Electrical and Electronics Engineers) комуникацията във всяка мрежа се осъществява на следните 7 нива:

#	ниво	описание и протоколи
7	Application (приложно ниво)	Осигурява на приложните програмисти интерфейс към мрежовата инфраструктура, осигурена от по-долните слоеве. Протоколите от това ниво задават форматите и правилата за обмяна на данни между комуникиращите приложения. Типични протоколи на това ниво са: HTTP, SMTP, POP3, FTP, SNMP, FTP, DNS, NFS и др.
6	Presentation (представително ниво)	Осигурява общ формат, унифицирано канонично представяне на пренасяните данни, което е еднакво за всички платформи и е разбираемо за по-долните слоеве. Типични протоколи или по-точно схеми за унифицирано представяне на данни от това ниво са XDR, ASN.1, SMB, AFP.
5	Session (сесийно ниво)	Организира и синхронизира прозрачната обмяна на информация между два процеса в операционните системи на комуникиращите машини. Типични протоколи от това ниво са: RPC, NetBIOS, CCITT X.225 и др.
4	Transport (транспортно ниво)	Осигурява поддръжката на комуникационни канали за данни между две машини. Позволява пренасяне не само на отделни пакети, но и на по-големи обеми данни. Осигурява прозрачност и надеждност на преноса на данни. Грижи се за започване, поддръжка и прекратяване на комуникацията между машините участнички. Типични протоколи на това ниво са:

		TCP, UDP, RTP, SPX, ATP.
3	Network (мрежово ниво)	Осигурява пренасяне на единици информация (пакети) между две машини в дадена мрежа, всяка от които има уникален мрежов адрес. Не е задължително двете машини да са пряко свързани една с друга и затова мрежовото ниво осигурява маршрутизиране на пакетите от една машина към друга с цел достигане на крайната цел. Типични протоколи на това ниво са IP, IPv6, ICMP, IGMP, X.25, IPX и др.
2	Data Link (свързващо ниво)	Осигурява директно пренасяне на информация между две мрежови комуникационни устройства (например две мрежови карти или два модема), управлява физическото ниво и се грижи за корекция на грешки възникнали в него. Типични протоколи са Ethernet, Token ring, PPP, Frame relay, ISDN и др.
1	Physical (физическо ниво)	Осигурява физическото пренасяне на информацията. Може да се реализира от радиовълни, оптични кабели, лазери и др.

Всяко от 7-те нива използва за работата си по-долните нива и не се интересува от това как точно те работят, а само от това какво те осигуряват. Това позволява на разработчиците да се абстрахират от ненужните детайли и да се концентрират върху работата само на нивата, които ги засягат. Така програмирането се опростява, защото програмистът не трябва да се съобразява с цялата сложност на комуникацията, а може да разчита на вече изградената инфраструктура от по-долните нива.

Пакет от протоколи TCP/IP

“TCP/IP protocol suite” не е протокол. TCP/IP е наименованието на пакета от протоколи, с които работи световната мрежа Интернет. В този пакет се включват протоколите IP, TCP, UDP, ICMP и IGMP. Локалните мрежи, работещи с протоколите от пакета TCP/IP се наричат Интранет мрежи.

4-слоен модел на TCP/IP мрежите

Класическият 7-слоен OSI модел засяга всички страни на организацията на комуникацията между две приложения, но често пъти с цел избягване на излишни детайли при Интернет и Интранет мрежи се използва опростен модел, т. нар. 4-слоен модел на TCP/IP мрежите. При него най-горните 3 слоя от OSI модела са обединени в един, защото

реално се отнасят до организацията на комуникацията на ниво приложни програми. Най-долните 2 слоя също са обединени, защото те заедно изпълняват една обща задача – осигуряват пренасянето на информация между две машини, които са директно свързани с някаква комуникационна линия. На практика TCP/IP моделът е опростен частен случай на OSI модела, при който на мрежово и транспортно ниво се използват протоколите от пакета TCP/IP. Приликите с OSI модела могат да се видят в таблицата с описанията на 4-те слоя:

#	ниво	описание и протоколи
4	Application (приложно ниво)	Осигурява на приложните програмисти интерфейс към мрежовата инфраструктура, осигурена от транспортния и Интернет слоевете. Протоколите от това ниво задават форматите и правилата за обмяна на данни между комуникиращите си приложения. Типични протоколи на това ниво са: HTTP, SMTP, POP3, FTP, SNMP, FTP, DNS, NFS и др.
3	Transport (транспортно ниво)	Осигурява поддръжката на комуникационни канали за данни между две приложения (евентуално на отдалечени машини). Позволява пренасяне не само на отделни пакети, но и на по-големи обеми данни. Осигурява прозрачност и надеждност на преноса. Грижи се за започване, поддръжка и прекратяване на комуникацията между процесите участници. В това ниво се използват само два протокола: TCP и UDP.
2	Internet (Интернет ниво)	Осигурява пренасяне на единици информация (пакети) между две машини в дадена мрежа, всяка от които има уникален адрес (IP адрес). Не е задължително двете машини да са пряко свързани една с друга и затова Интернет нивото осигурява маршрутизиране на пакетите от една машина към друга с цел достигане на крайната цел. На това ниво работят протоколите IP, IPv6, ICMP и IGMP.
1	Link (свързващо ниво)	Осигурява директно пренасяне на информация между две мрежови комуникационни устройства (например две мрежови карти или два модема). Типични протоколи са Ethernet, Token ring, PPP, Frame relay, ISDN и др.

Когато пишем Java програми, които комуникират по мрежата, ние програмираме най-горния слой от TCP/IP модела, така наречения Application слой. Преносът на данни, предизвикан от нашите Java програми, 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 и Интранет мрежи си има уникален IP адрес, по който можем да се обръщаме към него.

IP адреси

Основно понятие в Интернет и всички други TCP/IP мрежи е **IP адрес**. IP адресите представляват уникални 32-битови номера на компютри и се записват като четири 8-битови числа (в десетична бройна система), разделени по между си с точки. Всеки компютър, работещ в Интернет или Интранет мрежа, има IP адрес. Пример за IP адрес е записът: 212.39.1.17. Машините в TCP/IP базирани мрежи, които имат IP адрес, се наричат хостове (hosts).

Чрез проста сметка може да се прецени, че адресното пространство на Интернет се състои от около 4 милиарда IP адреса, но това не е съвсем така, защото няколко големи области от това пространство са резервирани за специални цели. Разпределението на IP адресното пространство на Интернет се управлява от световната организация IANA.

DNS

За улеснение на потребителите някои машини в Интернет освен IP адрес могат да имат и имена. Съответствията между IP адресите и имената на компютрите (хостовете в Интернет) се поддържат от специални **DNS сървъри**. При заявка DNS сървърите могат да намират IP адрес по име на машина и обратното. На едно име на хост в Интернет могат да съответстват няколко IP адреса, а също и на един IP адрес може да съответства повече от едно име.

Протоколът TCP

TCP (Transmission Control Protocol) е протокол, който осигурява надежден двупосочен комуникационен канал между две приложения. Можем да сравним този канал с канала, по който се осъществява при 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 (след определено време, наречено timeout) 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.

Протоколът UDP

UDP (User Datagram Protocol) 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 само изпращане на единични независими кратки съобщения (UDP пакети).

Портове – какво представляват и защо за необходими

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 и същ или няколко различни 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 едно от тях да бъдат обработени от съответното приложение, а не от някое друго. Така всяко приложение изпраща и получава своите данни независимо от другите, така сякаш те

не съществуват. Именно за решаване на този конфликт се използват портовете в протоколите TCP и UDP.

Портът е число между 0 и 65536 и задава уникален идентификатор на връзката в рамките на машината. Всеки TCP или UDP пакет, освен данните, които пренася, съдържа в себе си 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 4 числа: IP източник, порт източник, IP получател и порт получател.

Сокети

Сокет наричаме двойката (IP адрес; номер на порт). Комуникационният канал, който предоставя една TCP връзка наричаме **сокет връзка** (socket connection). Често пъти сокет връзките се наричат за краткост само **сокети**.

Как работят сокетите и портовете

Например нека нашият IP адрес е 212.50.1.81 и сме стартирали Internet Explorer и Outlook Express. С Internet Explorer браузваме някакъв сайт при което той е отворил няколко сокета към IP адрес 212.50.1.1 на порт 80 и тегли през тях някакви Web-страници и картинки. В същото време с Outlook Express си теглим новопристигналата поща 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. При отваряне на нова сокет връзка програмистът трябва да знае предварително IP адреса и порта на приложението, с който иска да осъществи комуникация.

Сървъри и клиенти

Съществуват два вида приложения, които комуникират по TCP протокола – клиентски и сървърски.

Клиентските приложения (наричани още **клиенти**) се свързват към сървърските като отварят сокет връзка към тях. За целта те предварително знаят техните IP адреси и портове.

Сървърските приложения (наричани още **сървъри**) "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.

Клиентите отварят сокети към сървърите, а сървърите създават сокети само по клиентска заявка, т.е. те не отварят сокети.

Програмен модел клиент/сървър

Можем да си представим едно клиент/сървър приложение 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 от вида „потребител на услуга/извършител на услуга".

Още за сокетите и портовете

Третата връзка от показаните по-горе свърза приложението Outlook Express, което се идентифицира с порт 1042 на нашата машина (212.50.1.81) 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, най-правилно е 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, който е уникален за цялата машина.

Портовете при UDP протокола

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.

Протоколи

Комуникационните канали, наречени сокети, не са достатъчни за осъществяване на комуникация между две приложения. Ако се върнем на ситуацията в магазина, клиентът трябва да комуникира с продавачката на известен и за двамата език. По същия начин при клиент/сървър комуникация клиентът и сървърът могат да си общуват само ако знаят един и същ език. Формални езици, които се използват за комуникация в компютърни мрежи, се наричат **протоколи**. Протоколите представляват системи от правила, които задават по какъв начин клиентът и сървърът могат да общуват и описват кои са валидните действия, които клиентът и сървърът могат да извършат във всеки един момент от комуникацията.

Услуги в Интернет и стандартни номера на портове

В Интернет работят много стандартни протоколи за комуникация между приложения, като всеки от тях е свързан с някаква **услуга**. Всяка услуга работи с някакъв протокол, предварително известен на клиентските и сървърските приложения. Например услугата достъп за Web-ресурси 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 (well-known services) има и асоциирани стандартни номера на портове (well-known ports), 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. Някои стандартни портове, протоколи и услуги са дадени в таблицата по-долу:

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

Класове за работа с мрежа в Java

Java приложенията могат да използват TCP и UDP протоколите за комуникация през Интернет чрез класовете от стандартния пакет **`java.net`**. Най-важните класове, който се използват при разработка на такива приложения са **`InetAddress`**, **`Socket`**, **`ServerSocket`**, **`DatagramSocket`**, **`DatagramPacket`** и **`URL`**.

По-нататък в тази глава ще разгледаме в детайли тези класове, но преди това ще направим кратък преглед на средствата за вход/изход и многонишково програмиране в Java, защото те са важна основа, без която не можем да създаваме мрежови приложения.

1.2. Вход/изход в Java

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.

Входно-изходни потоци

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.

Текстови потоци

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`**.

Четене от текстов поток

Най-важният метод от интерфейса **`java.io.Reader`** е методът **`read(…)`**, който служи за четене от текстов поток и се предоставя в няколко варианта съответно с различен набор от параметри:

`int read()` – прочита един символ и го връща във вид на число. Връща -1 ако е достигнат края на потока. Предизвиква **`IOException`** ако възникне грешка при четенето.
`int read(char[] cbuf)` – прочита поредица от символи и ги записва в подадения масив. Прочита най-много толкова символа, колкото е големината на масива. Връща броя на прочетените символи или -1 ако е достигнат края на потока. Предизвиква **`IOException`** ако възникне грешка при четенето.
`int read(char[] cbuf, int off, int len)` – прочита поредица от символи с максимална дължина **`len`** и ги записва в подадения масив на подаденото отместване **`off`**. Връща броя на прочетените символи или -1 ако е достигнат края на потока. Предизвиква **`IOException`** ако възникне грешка при четенето.

Всяка от изброените по-горе операции е блокираща, т.е. тя блокира при извикване и не връща управлението докато не прочете някакви данни

или не възникне входно-изходна грешка. Винаги при четене от текстови потоци е възможно да бъдат прочетени по-малко на брой символи, отколкото са заявени и това е една от важните особености, с които трябва да се съобразяваме, когато четем от текстови потоци.

Много удобен за четене от текстови потоци е класът **`java.io.BufferedReader`**, защото предлага метод за четене на цял текстов ред **`readLine()`**, а това често се налага.

Писане в текстов поток

Най-важният метод от интерфейса **`java.io.Writer`** е методът **`write(…)`**, който служи за писане в текстов поток. Той има няколко варианта:

`void write(int c)` – записва единичен символ в потока. Символът е представен като число. Предизвиква **`IOException`** ако възникне грешка при писането.
`void write(char[] cbuf)` – записва в потока последователността от символи, съдържаща се в подадения масив. Предизвиква **`IOException`** ако възникне грешка при писането.
`void write(char[] cbuf, int off, int len)` – записва в потока последователността от символи, съдържаща се в подадения масив, започваща от зададената позиция и имаща зададената дължина. Предизвиква **`IOException`** ако възникне грешка при писането.
`void write(String str)` – записва в потока даден символен низ. Предизвиква **`IOException`** ако възникне грешка при писането.

Както и при четенето от текстов поток всяка от изброените по-горе операции е блокираща, т.е. тя блокира при извикване и не връща управлението докато не запише данните или не възникне входно-изходна грешка. Операциите за писане в поток са или напълно успешни, т.е. записват всичките указани символи, или не са успешни и предизвикват изключение.

Важна операция при работа с текстови потоци е операцията **`flush()`**. Тя предизвиква реално изпращане на записаните данни към мястото, за което са предназначени, като се грижи за изпразване на всички буфери, използвани за кеширане на изпратените данни. Без да сме извикали **`flush()`** метода не можем да сме сигурни, че данните, които сме записали в даден поток с **`write(…)`**, наистина са отпътували към

местоназначението си. Когато разработваме приложения, които си комуникират чрез потоци, трябва винаги да внимаваме за тази особеност.

За писане в текстови потоци е удобен и класът **`java.io.PrintWriter`**, който има метод **`println(…)`** за печатане на цяла текстова линия.

Пример за работа с текстови потоци

Един прост пример за използване на текстови потоци е показан по-долу. Примерът представлява малка програмка, която номерира редовете на текстов файл:

TextFileLineNumberInserter.java

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

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

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

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

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

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.

Двоични потоци

Базов за всички входни двоични (бинарни) потоци е интерфейсът **`java.io.InputStream`**, а за всички изходни двоични потоци е

интерфейсът **java.io.OutputStream**. Ключов метод на **InputStream** е методът **int read(byte[] b, int off, int len)**, 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 към местоназначението ѝ. Методите **read(…)** и **write(…)** при двоичните потоци са напълно аналогични на съответните методи на текстовите потоци с разликата, че работят с двоични данни, а не със символи. Тези методи също са блокиращи, при четене също връщат броя прочетени байтове, който може да е по-малък от броя заявени за прочитане байтове, а при достигане на края на потока връщат -1, също хвърлят изключение при входно-изходна грешка и при писане също или се записва всичко, или се получава изключение.

За демонстрация на двоичните потоци ще дадем пример с една програмка, която копира двоични файлове:

BinaryFileCopier.java

```
import java.io.*;

public class BinaryFileCopier {
    public static void main(String args[]) throws IOException {
        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 == -1) break;
            outFile.write(buf, 0, bytesRead);
        }
        outFile.flush();
        outFile.close();
        inFile.close();
    }
}
```

1.3. Многонишково програмиране и синхронизация на нишки в Java

1.3.1. Многонишково програмиране в Java

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

Многонишкови програми

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 може 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). Много вероятно е освен това за всеки потребител на тази програма да се създава по още една нишка, за да може този потребител да се обслужва независимо от другите и да не бъде каран да чака, докато системата обслужва останалите.

Използване на нишки в Java

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, който илюстрира как чрез наследяване на класа **`Thread`** можем да създадем няколко нишки, които работят едновременно в рамките на нашето приложение:

ThreadTest.java

```
class MyThread extends Thread {
    private String mName;
    private long mTimeInterval;

    public MyThread(String aName, long aTimeInterval) {
        mName = aName;
        mTimeInterval = aTimeInterval;
    }

    public void run() {
        try {
            while (!isInterrupted()) {
                System.out.println(mName);
                sleep(mTimeInterval);
            }
        } catch (InterruptedException intEx) {
            // Current thread interrupted by another thread
        }
    }
}

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 програмка 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 следния:

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

Използване на интерфейса Runnable

Освен чрез наследяване на класа **`java.lang.Thread`** в Java можем да създаваме нишки и по друг начин – чрез имплементиране на интерфейса **`java.lang.Runnable`**. Начинът на работа е почти същия. Създаваме клас, който имплементира **`Runnable`** и пишем в метода му **`run()`** логиката на нишката. След това по този клас създаваме обект от класа **`Thread`** и му извикваме **`start()`** метода за стартиране на нишката. Класът **`Thread`** си има специален конструктор, който приема обекти имплементиращи **`Runnable`**.

Подходът с интерфейса **`Runnable`** се препоръчва да се използва тогава, когато поради някаква причина не можем да наследим класа **`Thread`**. Например ако нашият клас е вече наследник на някой друг клас, понеже в Java няма множествено наследяване, ако искаме да го изпълняваме в отделна нишка, нямаме друг избор освен да имплементираме **`Runnable`**.

Прекратяване изпълнението на нишки в Java

Досега разгледахме как можем да стартираме нова нишка. Често пъти освен да стартираме на нишки се налага и да спираме изпълнението на работещи нишки. Прекратяването на нишки има някои особености. То в никакъв случай не трябва да става насилствено чрез метода **`stop()`** на класа **`Thread`**. Вместо това нишката трябва учтиво да бъде помолена да прекрати работата си чрез извикване на метода й **`interrupt()`**. Затова по време на работата си всеки thread трябва от време на време да проверява, извиквайки метода **`isInterrupted()`**, дали не е помолен да прекрати работата си.

Други интересни методи на класа Thread са **`setPriority()`**, **`sleep()`** и **`setDaemon()`**, но за тях можем да прочетем повече документацията.

1.3.2. Синхронизация на нишки

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 достъп до общи ресурси при многонишковото програмиране.

Конфликти при едновременен достъп до общ ресурс

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 предположим, че сметките са обекти от класа **Account**, а операциите върху тях се извършват от класа **Bank**:

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

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

    double getAmmount() {
        return mAmmount;
    }
}

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 съвсем просто на следните три стъпки:

1) Прочита сумата от сметката.
2) Добавя сумата за внасяне към нея.
3) Записва новата сума в сметката.

Ако заявките за внасяне на пари от двамата клиента се изпълняват едновременно, може да се получи следният неприятен ефект:

1) Клиент 1 прочита сумата от сметката – 0 лева.
2) Клиент 2 прочита сумата от сметката – също 0 лева.
3) Клиент 1 прибавя към прочетената в стъпка 1) сума 100 лева и записва в сметката новата сума – 100 лева.
4) Клиент 2 прибавя към прочетената в стъпка 2) 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.

Синхронизация

Когато няколко конкурентни нишки или няколко отделни програми се опитват едновременно да осъществят достъп до общ ресурс, често пъти се получават неприятни ефекти подобни на този с банката. Такива ефекти наричаме **синхронизационни 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.

Средства за синхронизация в Java. Запазена дума synchronized

В Java средствата за синхронизация са вградени в самия език и са част от самата платформа. 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 aAcc, double aSum)
```

с декларацията

```
synchronized public static void deposit(
    Account aAcc, 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** на обекта при завършване на изпълнението на кода. 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 някакъв ред. 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. Препоръчително 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 могат 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.

Синхронизация с wait() и notify()

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 (произволна) от заспалите по даден обект нишки, а **notifyAll()** събужда всичките. Ако по обекта няма заспали нишки, **notify()** и **notifyAll()** не правят нищо.

Изчакване на ресурс и процесорно време

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.

Проблемът „производител-потребител”

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:

ProducerConsumerTest.java

```
import java.util.*;

class SharedQueue {
    private static final int QUEUE_SIZE = 5;
    private Vector mQueue = new Vector();
```

```
    public synchronized void put(String aObject)
            throws InterruptedException {
        while (mQueue.size() == QUEUE_SIZE)
            wait();
        mQueue.addElement(aObject);
        notify();
    }

    public synchronized Object get()
            throws InterruptedException {
        while (mQueue.size() == 0)
            wait();
        String message = (String) mQueue.firstElement();
        mQueue.removeElement(message);
        notify();
        return message;
    }
}

class Producer extends Thread {
    private SharedQueue mSharedQueue;

    public Producer(SharedQueue aSharedQueue) {
        mSharedQueue = aSharedQueue;
    }

    public void run() {
        try {
            while (true) {
                String message = new Date().toString();
                System.out.println("producer : put " + message);
                mSharedQueue.put(message);
                sleep(500);
            }
        } catch (InterruptedException e) {
        }
    }
}

class Consumer extends Thread {
    private SharedQueue mSharedQueue;

    public Consumer(SharedQueue aSharedQueue) {
        mSharedQueue = aSharedQueue;
    }

    public void run() {
        try {
            while (true) {
                String message =
                    (String) mSharedQueue.get();
                System.out.println(
```

```
                    getName() + " : get " + message);
                sleep(2000);
            }
        } catch (InterruptedException e) {
        }
    }
}

public class ProducerConsumerTest {
    public static void main(String args[]) {
        SharedQueue sharedQueue = new SharedQueue();
        Producer producer = new Producer(sharedQueue);
        producer.start();
        Consumer consumer1 = new Consumer(sharedQueue);
        consumer1.setName("consumer Mincho");
        consumer1.start();
        Consumer consumer2 = new Consumer(sharedQueue);
        consumer2.setName("consumer Pencho");
        consumer2.start();
    }
}
```

Как работи решението на проблема „производител-потребител”

Разглеждайки кода на горната програма 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. Това е така и в горната програма. На всякъде, където се извикват методите **`wait()`** и **`notify()`**, те са разположени в синхронизиран код.

Нека помислим малко какво се случва при изпълнението на горната програма. 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 ако първата нишка заспи по време на изпълнение на синхронизиран код, кодът, който я събужда, никога да не се изпълни, защото ще чака излизането на заспалата нишка от синхронизирания код. Изглежда, че и двете нишки ще заспят за вечни времена. Първата нишка ще чака втората да я събуди, а втората преди да се опита да събуди първата ще я чака да излезе от синхронизирания код за да изпълни своя синхронизиран код, а пък това няма да се случи никога понеже първата нишка е заспала. Изглежда, че има нещо нередно в горната програма.

Защо горните разсъждения са грешни? Отговорът ще открием ако се вгледаме внимателно в документацията на метода **`wait()`**. Извикването на **`wait()`** не само приспива текущата нишка, но и отключва обекта, по който тя е синхронизирана. Това позволява на блока, извикващ

notify(), който е синхронизиран по същия обект, да се изпълни без да чака. 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 deadlock съвсем не е изключено. Отговорност на програмиста е да предотврати възможността две или повече нишки в някой момент да започнат да се чакат взаимно.

В програмата по-горе най-интересните фрагменти са двата метода **put(…)** и **get()** на класа **SharedQueue**. Методът **put(…)** осигурява добавяне на съобщение в опашката. Ако опашката не е пълна, съобщението се добавя веднага и изпълнението на **put(…)** завършва веднага. Ако опашката, обаче, е пълна, методът **put(…)** блокира докато не се освободи място в нея, след което добавя съобщението и завършва изпълнението си. По същия начин работи и методът **get()**. При непразна опашка той се изпълнява веднага и изважда съобщението, което е наред, а ако опашката е празна, изчаква докато се напълни и след взима първото съобщение от нея. И двата метода **put(…)** и **get()** в края си извикват **notify()**, за да уведомят някоя от чакащите нишки, че нещо се е променило в състоянието на опашката, при което те евентуално биха могли да си свършат работата, за която чакат.

Ето какъв е приблизително резултата от изпълнението на горната програма:

```
producer : put Wed Mar 03 20:09:14 EET 2004
consumer Mincho : get Wed Mar 03 20:09:14 EET 2004
producer : put Wed Mar 03 20:09:14 EET 2004
consumer Pencho : get Wed Mar 03 20:09:14 EET 2004
producer : put Wed Mar 03 20:09:15 EET 2004
producer : put Wed Mar 03 20:09:15 EET 2004
consumer Mincho : get Wed Mar 03 20:09:15 EET 2004
producer : put Wed Mar 03 20:09:16 EET 2004
consumer Pencho : get Wed Mar 03 20:09:15 EET 2004
producer : put Wed Mar 03 20:09:16 EET 2004
producer : put Wed Mar 03 20:09:17 EET 2004
producer : put Wed Mar 03 20:09:17 EET 2004
consumer Mincho : get Wed Mar 03 20:09:16 EET 2004
producer : put Wed Mar 03 20:09:18 EET 2004
producer : put Wed Mar 03 20:09:18 EET 2004
consumer Pencho : get Wed Mar 03 20:09:16 EET 2004
```

```
producer : put Wed Mar 03 20:09:19 EET 2004
producer : put Wed Mar 03 20:09:19 EET 2004
...
```

Проблемът „производител-потребител” е много важен, дори основополагащ, при приложения, които използват сокети, защото в такива приложения често може да се получи ситуация, в която една нишка, която изпраща данни по някакъв сокет, трябва да чака друга нишка да прочете данните от някакъв друг източник (например също сокет). Имаме леко опростен вариант на класическия проблем – „потребител” (пишещата нишка), който чака „производителя” (четящата нишка).

1.4. 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`**. При разработка на сървъри обикновено трябва да се съобразяваме с необходимостта от обслужване на много потребители едновременно и независимо един от друг. Най-често този проблем се решава с използване на нишки за всеки потребител. Нека първо разгледаме по-простия вариант – обслужване само на един клиент в даден момент.

Прост TCP сървър

Да разгледаме сорс-кода на едно просто сървърско приложение – DateServer:

DateServer.java

```
import java.util.Date;
import java.io.OutputStreamWriter;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

public class DateServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(2002);
        while (true) {
        Socket socket = serverSocket.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 конструктора 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 връзка между сървъра и пристигналия клиент. От създадената сокет връзка 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. Сървърът 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:

```
Wed Mar 03 20:31:05 EET 2004
```

Прост TCP клиент

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. Ето как изглежда една примерна такава програмка:

DateServerClient.java

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

public class DateServerClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 2002);
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                socket.getInputStream() ) );
        System.out.println("The date on the server is: " +
            in.readLine());
        socket.close();
    }
}
```

Свързването към TCP 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. Ето какъв би могъл да е изхода от изпълнението на горната програмка, ако сървърът е стартиран на локалната машина и работи нормално:

```
The date on the server is: Wed Mar 03 20:34:12 EET 2004
```

Обработка на изключения

Ако стартираме клиентската програмка, но преди това спрем сървъра, ще получим малко по-различен резултат:

```
java.net.ConnectException: Connection refused: connect
    at java.net.PlainSocketImpl.socketConnect(Native Method)
    at java.net.PlainSocketImpl.doConnect(
        PlainSocketImpl.java:305)
    at java.net.PlainSocketImpl.connectToAddress(
        PlainSocketImpl.java:171)
    at java.net.PlainSocketImpl.connect(
        PlainSocketImpl.java:158)
    at java.net.Socket.connect(Socket.java:426)
    at java.net.Socket.connect(Socket.java:376)
    at java.net.Socket.<init>(Socket.java:291)
    at java.net.Socket.<init>(Socket.java:119)
    at DateServerClient.main(DateServerClient.java:6)
Exception in thread "main" Process terminated with exit code 1
```

Полученото изключение обяснява, че при опит за свързване към сървъра в конструктора на класа **`java.net.Socket`** се е получил проблем, защото съответният порт е бил затворен.

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 или в **`try`** ... **`catch`** блок или методът, в който се използва входно-изходна комуникация, да бъде обявен като метод, който може да породи изключението **`java.io.IOException`**. Изключения 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 при опит за писане в нея, ако сървърът се опита да слуша на зает вече порт, ако сървърът няма право да слуша на поискания порт, ако е изтекъл лимита от време за дадена блокираща операция и в много други случаи.

Четенето от сокет е блокираща операция

Една важна особеност при четенето от сокет е, че ако клиентът се опита да прочете данни от сървъра, а той не му изпрати нищо, клиентът ще блокира до затваряне на сокета, а при някои условия може да блокира дори за вечни времена (до спирането му). Затова сървърът и клиентът трябва да комуникират по предварително известен и за двамата протокол и да го спазват стриктно. Протоколът трябва да индикира по някакъв начин на клиента и на сървъра дали и кога да очакват получаването на още данни, както и колко данни да очакват.

Има няколко начина в един протокол да се укаже колко данни да очаква другата страна.

Единият начин е да се възприеме някой символ за край на изпращаните данни и отсрещната страна да чете от сокета докато не получи този символ. Най-често за такъв символ се възприема символът за край на ред. Много протоколи работят на принципа един текстов ред заявка, следван от един текстов ред отговор.

Другият начин е при изпращането на данни да се укаже първо дължината им в байтове, а след това да се изпратят толкова байта, колкото е указано. Така във всеки един момент отсрещната страна ще знае колко байта още й остават да прочете.

Писането във сокет също е блокираща операция

Трябва да се съобразяваме, че писането във сокет също е блокираща операция. Това означава, че ако се опитаме да изпратим някакви данни, не е гарантирано, че това няма да причини временно блокиране на текущата нишка. Обикновено при изпращането на някакво малко количество данни по сокет операцията не блокира, защото тези данни просто се прехвърлят в буфера за изпращане на изходния поток или в буфера за изпращане на сокета и реално не се изпращат докато не се извика **`flush()`** метода. При извикване на **`flush()`** метода също е възможно да не се получи блокиране и операцията да се изпълни без забавяне, но не винаги е така. Напълно е възможно при писане във сокет или при извикване на **`flush()`** да имаме забавяне от няколко секунди, дори и повече.

Какво става при прекъсване на връзката

Винаги, когато имаме комуникация по TCP сокет, е възможно във всеки един момент връзката между клиента и сървъра да бъде прекъсната по някаква причина. Например потребителят може да затвори внезапно

клиентското приложение или някой може да се спъне в мрежовия кабел. Възможно също е да спре внезапно тока или някоя машина просто да забие или да се изключи заради хардуерен проблем.

Има два вида прекъсване на връзката:

- нормален начин – чрез **`Socket.close()`** или чрез спиране на приложението – при него другата страна получава известяване, че сокетът е бил затворен;
- внезапен начин – при прекъсване на физическата свързаност между клиента и сървъра – при него никоя от страните не получава известяване и сокетът може да си остане отворен за вечни времена (ако не се вземат мерки).

Нормално прекъсване на TCP връзка

Да разгледаме какво се случва при нормално прекъсване на отворена TCP връзка съответно когато правим опит за четене или писане в нея.

Да разгледаме първо какво става при затваряне на сокет по време на четене от него. Нека имаме сървър, който очаква данни от клиента и е блокирал по операцията четене от сокет. Ако клиентът в даден момент прекрати връзката, например чрез метода **`close()`** на класа **`Socket`**, сървърът ще получи от клиента специален пакет (с вдигнат флаг FIN), по който ще разбере, че връзката се прекратява. Същото ще се получи и ако клиентското приложение внезапно бъде спряно. В този случай операционната система ще затвори всички сокети, свързани със завършилия процес като изпрати пакети до отсрещната страна за известяване на затварянето им.

За сървъра затварянето на сокета, от който той чете, означава край на входния поток, свързан с този сокет. В резултат на това сървърът ще излезе от блокираното си състояние и ще получи индикация за достигнат край на поток от прекъснатата операция за четене.

В някои случаи, когато сокетът бъде затворен насилствено вместо да се достигне края на потока, може да се получи изключение:

```
java.net.SocketException: Connection reset
```

Нека сега разгледаме какво се случва при затваряне на сокет по време на писане в него. Нека имаме сървър, който от време на време изпраща към клиента някакви данни по отворен TCP сокет. Ако в даден момент клиентът затвори сокета, сървърът ще получи специален FIN пакет, който известява затварянето и ще си отбележи в обекта, свързан със съответния сокет, че той вече е затворен. При последващ опит за писане

в изходния потока, свързан със затворения сокет, ще се получи изключението:

```
java.net.SocketException: Connection reset by peer: socket write
error
```

Внезапно прекъсване на TCP връзка

При внезапното (аварийно) прекъсване на дадена TCP връзка нещата стоят по-различно. Да предположим, че клиент и сървър си говорят по отворен TCP сокет. В даден момент връзката между тях се разпада, например заради физическо прекъсване на кабела, който ги свързва. От този момент нататък по отворения сокет нито клиентът нито сървърът ще получи някакви данни, но сокетът няма да се затвори.

Ако някоя от страните се опита да изпрати нещо по този сокет, след известно време (някакъв timeout) ще получи изключението:

```
java.net.SocketException: Connection reset by peer: socket write
error
```

Ако някоя от страните е блокирала по четене от този сокет, има проблем. Тя никога няма да разбере, че връзката се е разпаднала, защото няма да получи пакет, който да съобщава това (понеже линията е разрушена). Дори операционната система няма да разбере, че сокетът е невалиден. Ако напишем командата "**`netstat`**", можем да видим, че дори и след няколко часа сокетът ще си стои в състояние „отворен", въпреки че връзката реално е прекратена.

Описаният сценарий може да породи много сериозни проблеми за един TCP сървър. Ако от време на време клиенти се свързват към сървъра и връзката им се разпада, а сървърът никога не разбира за това, след няколко дни или седмици ресурсите на сървъра ще свършат и той ще спре да работи или ще започне да се държи неадекватно.

Как да установяваме прекъсната TCP връзка

Има няколко препоръки, които трябва да спазваме, ако не искаме да попаднем в ситуация, в която безкрайно дълго се опитваме да четем от невалиден (разрушен) TCP сокет, както в описания сценарий.

Едната препоръка е никога да не четем от сокет без ограничение откъм време. В класа **`java.net.Socket`** има метод **`setSoTimeout(int)`**, с който може да се задава максималното време в милисекунди, за което една операция **`read()`** от **`InputStream`**-а, свързан с даден сокет трябва да приключи. Ако за зададеното време по сокета не пристигне нищо, се

поражда изключението **`java.net.SocketTimeoutException`** и операцията четене се прекратява. По подразбиране стойността, зададена в **`setSoTimeout(int)`** е 0, което означава, че ограничение във времето няма.

Подходът със задаване на timeout при четене от сокет решава проблема с безкрайното чакане на данни от невалиден сокет, но не винаги е подходящ.

Понякога е възможно един сокет да е валиден, но по него да не преминават никакви данни в продължение на часове. Например, ако имаме сървър, който приема някаква информация от клиентите си от време на време, когато някой клиент реши да му изпрати нещо, е възможно с часове нищо да не бъде изпратено нито от клиента към сървъра, нито в обратната посока. Въпреки продължителната липса на активност, връзката не трябва да се прекъсва след изминаване на някакъв timeout (примерно 1, 5 или 10 минути). Сървърът, обаче иска ако се случи нещо с клиента и връзката с него се разпадне внезапно, да разбере за това и да освободи ресурсите, отделени за обслужването на този клиент.

Основният проблем е, че при липса на трафик по даден сокет няма начин да се провери дали връзката е активна или е разрушена.

Най-добрият начин да се справим с този проблем е да реализираме разширение на протокола, което осигурява възможност за изпращане на проверяващи пакети от сървъра към клиента от време на време, примерно на 2-3 минути. Така по сокета през определено време ще преминават някакви данни и ако връзката е прекъсната, ще настъпва изключение и сървърът ще разбира, че клиентът е недостъпен.

Този подход е най-надеждният и най-сигурният, но може да причини значителни усложнения при разработката на клиента и сървъра заради синхронизационни проблеми, защото проверяващите пакети трябва да се изпращат и обработват отделно и независимо от другите пакети.

Има и друг вариант да се справим с проблема. Той не изисква промяна на протокола, но и не е толкова надежден. TCP сокетите имат стандартна възможност да бъдат автоматично проверявани през определено време дали са свързани чрез специални keep-alive пакети, на които отсрещната страна е длъжна да отговаря. Тази възможност се поддържа вътрешно от TCP протокола и операционната система и ако бъде включена, при липса на отговори на тези keep-alive пакети за определено време (някакъв системен timeout, който обикновено е 2 часа), се счита, че сокетът е невалиден. Класът **`Socket`** в Java има метод

`setKeepAlive(boolean)`, с който се задава дали да бъде включена keep-alive опцията за даден сокет. По подразбиране тази опция е изключена. Проблемът на този подход е, че не знаем със сигурност колко е keep-alive timeout стойността и за различните платформи тя е различна. Обикновено стойността е няколко часа, което означава, че при сриване на връзката сървърът ще разбере за това не веднага, а едва след няколко часа.

Ако при четене от сокет, за който е зададена keep-alive опцията, се установи, че връзката се е разпаднала, в нишката, която е блокирала по операцията четене, се предизвиква изключението:

```
java.net.SocketException: Connection reset
```

Като правило, ако не искаме да попаднем в ситуация, в която безкрайно дълго се опитваме да четем от невалиден сокет, трябва или да имаме ограничение на максималното време за четене (timeout) или трябва да реализираме изпращането на проверяващи данни от време на време, или поне трябва да включваме keep-alive опцията на сокета, от който четем.

Обслужване на много потребители едновременно

Даденият по-горе пример за сървър обслужва клиентите си последователно един след друг. 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 изчакват. Затова се налага сървърът да обслужва клиентите си едновременно и независимо един от друг. За реализация на такава стратегия в средата на Java най-често се използва многонишковият 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. Ако трябва да сме точни, от JDK 1.4 в Java се поддържат и асинхронни сокети, с които могат да се обработват едновременно много клиенти само с една нишка, но засега няма да разглеждаме този програмен модел.

Многопотребителски сървър-речник

Да си поставим за задача реализацията на прост сървър, който по зададена дума на английски език връща преводът ѝ на български език, а за при непозната думи връща грешка. Сървърът трябва да може да

обслужва много потребители едновременно и независимо един от друг, без да е необходимо някой от тях да чака докато сървърът обслужва останалите. За простота ще считаме, че думите и техните преводи са дадени като константа с ясната идея, че в една реална ситуация те трябва да се извличат от база данни или от някаква друга система. Ето как можем да реализираме нашият сървър-речник:

DictionaryServer.java

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;

public class DictionaryServer {
    public static int LISTENING_PORT = 3333;

    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket =
            new ServerSocket(LISTENING_PORT);
        System.out.println("Server started.");
        while (true) {
            Socket socket = serverSocket.accept();
            DictionaryClientThread dictionaryClientThread =
                new DictionaryClientThread(socket);
            dictionaryClientThread.start();
        }
    }
}

class DictionaryClientThread extends Thread {
    private int CLIENT_REQUEST_TIMEOUT = 15*60*1000; // 15 min.
    private Socket mSocket;
    private BufferedReader mSocketReader;
    private PrintWriter mSocketWriter;

    public DictionaryClientThread(Socket aSocket)
            throws IOException {
        mSocket = aSocket;
        mSocket.setSoTimeout(CLIENT_REQUEST_TIMEOUT);
        mSocketReader = new BufferedReader(
            new InputStreamReader(mSocket.getInputStream()));
        mSocketWriter = new PrintWriter(
            new OutputStreamWriter(mSocket.getOutputStream()));
    }

    public void run() {
        System.out.println(new Date().toString() + " : " +
            "Accepted client : " + mSocket.getInetAddress() +
            ":" + mSocket.getPort());
        try {
```

```
            mSocketWriter.println("Dictionary server ready.");
            mSocketWriter.flush();
            while (!isInterrupted()) {
                String word = mSocketReader.readLine();
                if (word == null)
                    break; // Client closed the socket
                String translation = getTranslation(word);
                mSocketWriter.println(translation);
                mSocketWriter.flush();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        System.out.println(new Date().toString() + " : " +
            "Connection lost : " + mSocket.getInetAddress() +
            ":" + mSocket.getPort());
    }

    private String getTranslation(String aWord) {
        if (aWord.equalsIgnoreCase("network")) {
            return "мрежа";
        } else if (aWord.equalsIgnoreCase("firewall")) {
            return "защитна стена";
        } else {
            return "! непозната дума !";
        }
    }
}
```

Как работи сървърът-речник

Сървърът-речник е изключително прост. Той отваря за слушане сървърски сокет на порт 3333 и започва да слуша в цикъл за клиентски заявки идващи към този сокет. При приемане на клиент създава нишка, която да го обслужва, подава й създадения клиентски сокет и я стартира.

Нишката, която обслужва клиентите, първо им изпраща поздравително съобщение, след което в цикъл чете дума от клиента, намира преводът й в речника си и изпраща към клиента този превод. Това продължава докато нишката не бъде помолена да завърши работата си. Ако междувременно от клиента се прочете празен низ, това означава, че е достигнат края на входния поток, т.е. клиентът е затворил сокет връзката. В такъв случай се прекратява обслужването на клиента и нишката завършва.

За клиентския сокет се задава timeout при четене 15 минути. Това се прави с цел да се прекъсват автоматично връзките на клиентите, които по някаква причина много дълго бездействат. Такива клиенти могат или

да са изгубили по някаква причина връзката със сървъра или просто да са свързани, но да не са активни. И в двата случая е добре сървърът да ги премахва, за да не хаби излишни ресурси.

Забележете, че веднага след като изпратим нещо към сървъра извикваме **`flush()`** метода, за да осигурим реалното изпращане на данните по сокета. Ако не извикаме **`flush()`**, данните ще останат да чакат в буфера на класа **`PrintWriter`** и няма да отпътуват по сокета, все едно не са изпратени към потока. Тази особеност с буферирането е много важна при комуникация с потоци и винаги трябва да се съобразяваме с нея.

Клиент за сървъра-речник

Нека сега напише и клиент за нашия сървър речник. Всичко, което трябва да прави клиента е да се свърже със сървъра, да извлече от него поздравителното съобщение и след това постоянно да чете дума от конзолата, да я изпраща към сървъра за превод и да отпечатва превода получен от клиента. Ето една реализация:

DictionaryClient.java

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

public class DictionaryClient {
    private static int SERVER_RESPONSE_TIMEOUT = 60*1000;
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 3333);
        socket.setSoTimeout(SERVER_RESPONSE_TIMEOUT);
        BufferedReader socketReader = new BufferedReader(
            new InputStreamReader(socket.getInputStream()) );
        PrintWriter socketWriter =
            new PrintWriter(socket.getOutputStream());
        BufferedReader consoleReader = new BufferedReader(
            new InputStreamReader(System.in) );
        String welcomeMessage = socketReader.readLine();
        System.out.println(welcomeMessage);
        try {
            while (true) {
                String word = consoleReader.readLine();
                socketWriter.println(word);
                socketWriter.flush();
                String translation = socketReader.readLine();
                System.out.println(translation);
            }
        } finally {
            socket.close();
        }
    }
```

```
}
```

Как работи клиентът за сървъра-речник

Всичко което прави клиентът е да отвори сокет към сървъра, да прочете от него поздравителното съобщение, след което в безкраен цикъл да чете дума от конзолата, да я изпраща към сървъра за превод, да прочита отговора на сървъра и да го отпечатва в конзолата. Забележете отново, че след изпращане на заявката към сървъра се извиква методът **`flush()`** на изходния поток. Ако този метод не се извика, програмата ще блокира, защото заявката няма да достигне сървъра. Ако ги няма ограниченията за максимално допустимо време за четене на сървъра и на клиента, програмата ще се опитва неограничено дълго време да прочете отговора на сървъра, а сървърът ще чака неограничено дълго време да получи заявка от клиента и така никой няма да дочака другия. В случая максималното допустимо време за чакане на отговор от сървъра се ограничава до 1 минута веднага след успешно свързване към сървъра.

1.4.1. TCP forward сървър

Вече знаем как да разработваме многопотребителски TCP сървъри. Сега ще си поставим малко по-сложна задача – разработка на сървър за препращане на трафика от един TCP порт към друг TCP порт на друга машина по прозрачен за потребителя начин. Такъв софтуер се нарича bridge на транспортно ниво.

Какво всъщност прави един TCP forward сървър

Представете си, че имаме локална мрежа с локални IP адреси `192.168.0.*`, която е свързана с Интернет през една машина с реален IP адрес от Интернет (статичен IP адрес), да кажем `212.50.1.1`. От Интернет се вижда само една машина от цялата мрежа – машината `212.50.1.1`, а всички останали машини от мрежата не са достъпни, защото нямат реален IP адрес в Интернет. Искаме да пуснем някакъв TCP сървър (някаква услуга), да кажем на порт `80` на някоя машина от локалната мрежа, да кажем `192.168.0.12` и искаме тази услуга да е достъпна от Интернет. Ако просто стартираме TCP сървъра, услугата ще е достъпна само за потребителите на локалната мрежа.

Има няколко варианта да накараме услугата да е достъпна и от Интернет. Най-лесният от тях е да си осигурим реален IP адрес за машината, на която работи сървъра, но това не винаги е възможно и може да изисква допълнителни разходи.

Друг вариант е да се направи т. нар. **port forwarding** (препращане на порт) на някой порт от машината `212.50.1.1` към някой порт на машината `192.168.0.12`. Целта е всеки, който се свърже към `212.50.1.1` на даден порт за препращане да получава на практика връзка към `192.168.0.12` на порт 80. Има различни програми, които извършват препращане на порт, някои от които се разпространяват стандартно с мрежовия софтуер на операционната система.

Нашата цел е да напишем програма на Java, която извършва **TCP port forwarding**.

Примерен TCP forward сървър

Нашият сървър трябва да слуша на даден TCP порт и при свързване на клиент да отваря сокет към дадена машина на даден порт (сървъра) и да осигурява препращане на всичко идващо от клиента към сървъра и на всичко, идващо от сървъра към клиента. При прекъсване на връзката с клиента трябва да се прекъсне и връзката със сървъра и обратното – при

прекъсване на връзката със сървъра трябва да се прекъсне и връзката с клиента. Трябва да се поддържа обслужване на много потребители едновременно и независимо един от друг. Ето една примерна реализация на такъв TCP forward сървър:

TCPForwardServer.java

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

/**
 * TCPForwardServer is a simple TCP bridging software that
 * allows a TCP port on some host to be transparently forwarded
 * to some other TCP port on some other host. TCPForwardServer
 * continuously accepts client connections on the listening TCP
 * port (source port) and starts a thread (ClientThread) that
 * connects to the destination host and starts forwarding the
 * data between the client socket and destination socket.
 */
public class TCPForwardServer {
    public static final int SOURCE_PORT = 2525;
    public static final String DESTINATION_HOST = "mail.abv.bg";
    public static final int DESTINATION_PORT = 25;

    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket =
            new ServerSocket(SOURCE_PORT);
        while (true) {
            Socket clientSocket = serverSocket.accept();
            ClientThread clientThread =
                new ClientThread(clientSocket);
            clientThread.start();
        }
    }
}

/**
 * ClientThread is responsible for starting forwarding between
 * the client and the server. It keeps track of the client and
 * servers sockets that are both closed on input/output error
 * durinf the forwarding. The forwarding is bidirectional and
 * is performed by two ForwardThread instances.
 */
class ClientThread extends Thread {
    private Socket mClientSocket;
    private Socket mServerSocket;
    private boolean mForwardingActive = false;

    public ClientThread(Socket aClientSocket) {
        mClientSocket = aClientSocket;
    }
```

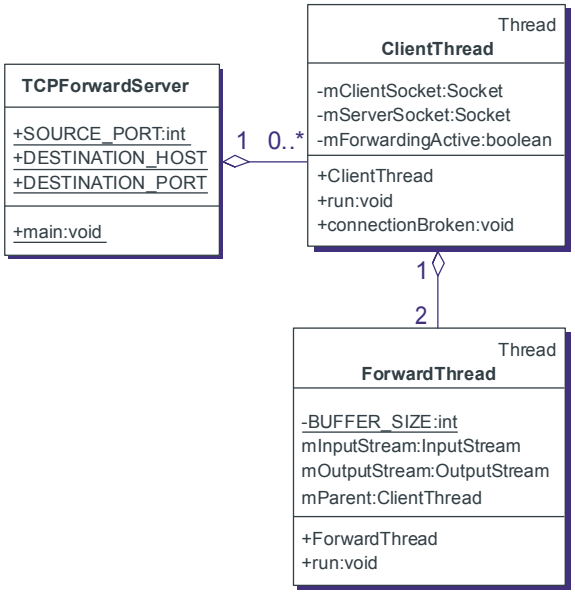
```
    /**
     * Establishes connection to the destination server and
     * starts bidirectional forwarding ot data between the
     * client and the server.
     */
    public void run() {
        InputStream clientIn;
        OutputStream clientOut;
        InputStream serverIn;
        OutputStream serverOut;
        try {
            // Connect to the destination server
            mServerSocket = new Socket(
                TCPForwardServer.DESTINATION_HOST,
                TCPForwardServer.DESTINATION_PORT);

            // Turn on keep-alive for both the sockets
            mServerSocket.setKeepAlive(true);
            mClientSocket.setKeepAlive(true);

            // Obtain client & server input & output streams
            clientIn = mClientSocket.getInputStream();
            clientOut = mClientSocket.getOutputStream();
            serverIn = mServerSocket.getInputStream();
            serverOut = mServerSocket.getOutputStream();
        } catch (IOException ioe) {
            System.err.println("Can not connect to " +
                TCPForwardServer.DESTINATION_HOST + ":" +
                TCPForwardServer.DESTINATION_PORT);
            connectionBroken();
            return;
        }

        // Start forwarding data between server and client
        mForwardingActive = true;
        ForwardThread clientForward =
            new ForwardThread(this, clientIn, serverOut);
        clientForward.start();
        ForwardThread serverForward =
            new ForwardThread(this, serverIn, clientOut);
        serverForward.start();

        System.out.println("TCP Forwarding " +
            mClientSocket.getInetAddress().getHostAddress() +
            ":" + mClientSocket.getPort() + " <--> " +
            mServerSocket.getInetAddress().getHostAddress() +
            ":" + mServerSocket.getPort() + " started.");
    }

    /**
```

```
     * Called by some of the forwarding threads to indicate
     * that its socket connection is brokean and both client
     * and server sockets should be closed. Closing the client
     * and server sockets causes all threads blocked on reading
     * or writing to these sockets to get an exception and to
     * finish their execution.
     */
    public synchronized void connectionBroken() {
        try {
            mServerSocket.close();
        } catch (Exception e) {}
        try {
            mClientSocket.close(); }
        catch (Exception e) {}

        if (mForwardingActive) {
            System.out.println("TCP Forwarding " +
                mClientSocket.getInetAddress().getHostAddress()
                + ":" + mClientSocket.getPort() + " <--> " +
                mServerSocket.getInetAddress().getHostAddress()
                + ":" + mServerSocket.getPort() + " stopped.");
            mForwardingActive = false;
        }
    }
}

/**
 * ForwardThread handles the TCP forwarding between a socket
 * input stream (source) and a socket output stream (dest).
 * It reads the input stream and forwards everything to the
 * output stream. If some of the streams fails, the forwarding
 * stops and the parent is notified to close all its sockets.
 */
class ForwardThread extends Thread {
    private static final int BUFFER_SIZE = 8192;

    InputStream mInputStream;
    OutputStream mOutputStream;
    ClientThread mParent;

    /**
     * Creates a new traffic redirection thread specifying
     * its parent, input stream and output stream.
     */
    public ForwardThread(ClientThread aParent, InputStream
            aInputStream, OutputStream aOutputStream) {
        mParent = aParent;
        mInputStream = aInputStream;
        mOutputStream = aOutputStream;
    }

    /**
```

```
     * Runs the thread. Continuously reads the input stream and
     * writes the read data to the output stream. If reading or
     * writing fail, exits the thread and notifies the parent
     * about the failure.
     */
    public void run() {
        byte[] buffer = new byte[BUFFER_SIZE];
        try {
            while (true) {
                int bytesRead = mInputStream.read(buffer);
                if (bytesRead == -1)
                    break; // End of stream is reached --> exit
                mOutputStream.write(buffer, 0, bytesRead);
                mOutputStream.flush();
            }
        } catch (IOException e) {
            // Read/write failed --> connection is broken
        }

        // Notify parent thread that the connection is broken
        mParent.connectionBroken();
    }
}
```

Как работи примерният TCP forward сървър

Сървърът се състои от няколко класа, които са видими от диаграмата:



Главната програма е доста проста. Тя слуша постоянно за идващи заявки на TCP порт 2525 и при свързване на нов клиент създава нишка от класа **ClientThread**, подава й сокета, създаден за този клиент и стартира нишката.

Класът **ClientThread** се опитва да се свържи към сървъра (в случая това е хоста **mail.abv.bg** на порт **25**, където работи стандартната услуга за изпращане на поща по протокол SMTP). При успешно свързване към сървъра се създават още две нишки **ForwardThread**. Едната нишка транспортира всичко получено от сокета на клиента към сокета на сървъра, а другата нишка транспортира всичко получено от сокета на сървъра към клиента. При неуспешно свързване към сървъра сокетът на клиента се затваря.

Нишката **ForwardThread** не е сложна. тя се създава по два потока – един входен и един изходен. Всичко, което тя прави, е да чете от входния поток и да пише в изходния поток. При достигане на края на входния поток или при възникване на входно-изходна грешка се извиква специален метод на **ClientThread** класа, с който се спира препращането на трафика между клиента и сървъра и нишката завършва изпълнението си.

Транспортирането на информация се извършва на бинарно ниво, защото това е единственият правилен начин. Ако се използваха текстови потоци, щеше да има проблеми ако клиентът и сървърът използват бинарен протокол.

ClientThread нишката съществува само докато се свърже към сървъра и стартира препращащите нишки (**ForwardThread**), след което завършва изпълнението си.

Ако в някоя от препращащите нишки се установи проблем с препращането, това означава, че се е прекъснала връзката съответно или с клиента или със сървъра. В такъв случай и двете нишки за препращане на данни между клиента и сървъра трябва да завършат. Това се осигурява чрез затваряне на двата сокета, по които върви комуникацията. Затварянето на активен сокет предизвиква изключение в нишката, която е блокирала по операцията четене или писане в този сокет, което осигурява прекъсване на изпълнението и на другата препращащата нишка, която все още е активна.

На практика ако сървърът затвори сокета, се затваря и сокета на клиента и всички нишки, свързани с обслужването на този клиент прекратяват изпълнението си. Ако клиентът затвори сокета, се затваря и сокета към сървъра и също всички нишки, свързани с този клиент приключват.

За да не се получава ситуация, при което TCP forward сървърът е загубил връзката или със сървъра или с клиента и чака безкрайно дълго да прочете данни от тях, за използваните сокети се задава опцията keep-alive и така при изгубване на връзката с някоя от страните, най-късно след няколко часа forward сървърът ще разбере и ще затвори връзката и с другата страна. Това е единственият начин да се контролират изгубените връзки, защото TCP forward сървърът няма никаква информация за протокола, по който ги говорят клиента и сървъра.

TCP forward сървърът в действие

Ето какъв изход би могъл да се получи ако при активен **`TCPForwardServer`** се свържем към него на порт 2525 и напишем няколко команди към SMTP сървъра:

```
telnet localhost 2525
220 abv.bg ESMTP
HELO
250 abv.bg
HELP
214 netqmail home page: http://qmail.org/netqmail
QUIT
221 abv.bg

Connection to host lost.
```

Ето и изходът на конзолата на сървъра след изпълнението на горните команди:

```
TCP Forwarding 127.0.0.1:4184 <--> 194.153.145.80:25 started.
TCP Forwarding 127.0.0.1:4184 <--> 194.153.145.80:25 stopped.
```

Няма никаква съществена разлика дали се свързваме директно към **`mail.abv.bg`** на порт **`25`** или към **`localhost`** на порт **`2525`**. Това беше и целта на TCP forward сървъра – да осигури прозрачно препращане на някой TCP порт.

Има само един малък проблем. Ако **`mail.abv.bg`** по някаква причина не работи вместо да се получи съобщение за отказана връзка:

```
telnet mail.abv.bg 25
Connecting To mail.abv.bg...Could not open connection to the
host, on port 25: Connect failed
```

се осъществява успешно свързване към localhost:2525, след което сокетът се затваря. Правилното поведение би било въобще да се откаже свързване към TCP forward сървъра.

Проблемът идва от това, че нашият сървър винаги приема клиентски заявки независимо дали сървърът е готов и може също да приема клиентски заявки. При по-добрите port forward сървъри нямат такъв дефект, но те обикновено работят на по-ниско ниво. Този дефект може да се преодолее чрез използване на асинхронни сокети, които се поддържат в Java от версия 1.4, но ние няма да се занимаваме с това.

1.4.2. Многопотребителски сървър за разговори (chat server)

Нека сега си поставим една още по-сложна задача – реализация на сървър за разговори (chat server). Чрез него ще демонстрираме в пълнота силата на многонишковото програмиране при разработка на мрежови приложения. Да разгледаме първо една примерна реализация на многопотребителски сървър за разговори:

ChatServer.java

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

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

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

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

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

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

```
    public void run() {
        try {
            while (!isInterrupted()) {
                String msg = mSocketReader.readLine();
                if (msg == null)
                    break;
                mDispatcher.dispatchMsg(mSocket, msg);
            }
        } catch (IOException ioex) {
            System.err.println("Error communicating " +
                "with some of the clients.");
        }
        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) {
                // Probably the socket already is closed
            }
        }
    }

    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) {
            // Thread interrupted. Do nothing
        }
    }
}
```

Как работи сървърът за разговори

Като функционалност сървърът не е много сложен. Единственото, което прави, 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 (**`ChatServer`**), 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. При приемане на нов клиент той първо се добавя в списъка на диспечера, а след това се създава една нишка за получаване на съобщенията идващи от него и тази нишка се 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 вика **`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 приложение на модела "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 потребител (консуматор) на съобщения. Ако опашката е празна, нишката чака (като извиква **`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 (т.е. когато връзката със клиента се разпадне). Така например, ако клиентът затвори сокета, той ще бъде премахнат от списъка, защото четенето на съобщение от него ще се провали.

Защо сървърът за разговори не е добре написан

Макар и всичко да изглежда добре, в този сървър има един сериозен проблем. Сървърът наистина е способен да обслужва много клиенти едновременно, но не може все още да твърди, че е добре написан. Проблемът е, че нишката-диспечер, която изпраща получените съобщения, работи последователно с един for-цикъл. Тя не преминава към изпращането на следващо съобщение от опашката докато не изпрати текущото съобщение на всички клиенти. Ако връзката с един от клиентите е много бавна, заради него ще се наложи всички да чакат преди да получат следващото изпратено съобщение. Следователно е необходимо диспечерът да разпраща съобщенията от опашката в някакъв смисъл паралелно.

Как може да се подобри chat сървърът

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, памет и други ресурси (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.

Има и по-разумен вариант – при свързването на нов клиент за него да се създава още една нишка, която служи за разпращане на съобщенията, предназначени конкретно 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 опашката е празна и да се събужда, когато в нея постъпи съобщение, за да започне изпращането му. Тази нишка може да се реализира по същия начин като класа **`ServerMsgDispatcher`**, защото има много сходна функционалност. Тя трябва само да чака в опашката й да постъпят някакви съобщения, след това да ги разпраща едно по едно. Да видим как можем да реализираме описаната идея.

Разработка на истински многопотребителски chat сървър

NakovChatServer.java

```
/**
 * Nakov Chat Server
 * (c) Svetlin Nakov, 2002
 * http://www.nakov.com
 *
 * Nakov Chat Server is multithreaded chat server. It accepts
 * multiple clients simultaneously and serves them. Clients are
 * able to send messages to the server. When some client sends
 * a message to the server, the message is dispatched to all
 * the clients connected to the server.
 *
 * The server consists of two components - "server core" and
 * "client handlers".
 *
 * The "server core" consists of two threads:
 *   - NakovChatServer - accepts client connections, creates
 * client threads to handle them and starts these threads
 *   - ServerDispatcher - waits for messages and when some
 * message arrive sends it to all the clients connected to
 * the server
 *
 * The "client handlers" consist of two threads:
 *   - ClientListener - listens for message arrivals from the
 * socket and forwards them to the ServerDispatcher thread
```

```
 *   - ClientSender - sends messages to the client
 *
 * For each accepted client, a ClientListener and ClientSender
 * threads are created and started. A Client object is also
 * created to maintain the information about the client and is
 * added to the ServerDispatcher's clients list. When some
 * client is disconnected, is it removed from the clients list
 * and both its ClientListener and ClientSender threads are
 * interrupted.
 */

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

/**
 * NakovChatServer class is the entry point for the server.
 * It opens a server socket, starts the dispatcher thread and
 * infinitely accepts client connections, creates threads for
 * handling them and starts these threads.
 */
public class NakovChatServer {
    public static final int LISTENING_PORT = 2002;
    public static String KEEP_ALIVE_MESSAGE = "!keep-alive";
    public static int CLIENT_READ_TIMEOUT = 5*60*1000;
    private static ServerSocket mServerSocket;

    private static ServerDispatcher mServerDispatcher;

    public static void main(String[] args) {
        // Start listening on the server socket
        bindServerSocket();

        // Start the ServerDispatcher thread
        mServerDispatcher = new ServerDispatcher();
        mServerDispatcher.start();

        // Infinitely accept and handle client connections
        handleClientConnections();
    }

    private static void bindServerSocket() {
        try {
            mServerSocket = new ServerSocket(LISTENING_PORT);
            System.out.println("NakovChatServer started on " +
                "port " + LISTENING_PORT);
        } catch (IOException ioe) {
            System.err.println("Can not start listening on " +
                "port " + LISTENING_PORT);
            ioe.printStackTrace();
            System.exit(-1);
        }
```

```
    }

    private static void handleClientConnections() {
        while (true) {
            try {
                Socket socket = mServerSocket.accept();
                Client client = new Client();
                client.mSocket = socket;
                ClientListener clientListener = new
                    ClientListener(client, mServerDispatcher);
                ClientSender clientSender =
                    new ClientSender(client, mServerDispatcher);
                client.mClientListener = clientListener;
                clientListener.start();
                client.mClientSender = clientSender;
                clientSender.start();
                mServerDispatcher.addClient(client);
            } catch (IOException ioe) {
                ioe.printStackTrace();
            }
        }
    }
}


/**
 * ServerDispatcher class is purposed to listen for messages
 * received from the clients and to dispatch them to all the
 * clients connected to the chat server.
 */
class ServerDispatcher extends Thread {
    private Vector mMessageQueue = new Vector();
    private Vector mClients = new Vector();

    /**
     * Adds given client to the server's client list.
     */
    public synchronized void addClient(Client aClient) {
        mClients.add(aClient);
    }

    /**
     * Deletes given client from the server's client list if
     * the client is in the list.
     */
    public synchronized void deleteClient(Client aClient) {
        int clientIndex = mClients.indexOf(aClient);
        if (clientIndex != -1)
            mClients.removeElementAt(clientIndex);
    }

    /**
```

```
     * Adds given message to the dispatcher's message queue and
     * notifies this thread to wake up the message queue reader
     * (getNextMessageFromQueue method). dispatchMessage method
     * is called by other threads (ClientListener) when a
     * message is arrived.
     */
    public synchronized void dispatchMessage(
            Client aClient, String aMessage) {
        Socket socket = aClient.mSocket;
        String senderIP =
            socket.getInetAddress().getHostAddress();
        String senderPort = "" + socket.getPort();
        aMessage = senderIP + ":" + senderPort +
            " : " + aMessage;
        mMessageQueue.add(aMessage);
        notify();
    }

    /**
     * @return and deletes the next message from the message
     * queue. If there is no messages in the queue, falls in
     * sleep until notified by dispatchMessage method.
     */
    private synchronized String getNextMessageFromQueue()
    throws InterruptedException {
        while (mMessageQueue.size()==0)
            wait();
        String message = (String) mMessageQueue.get(0);
        mMessageQueue.removeElementAt(0);
        return message;
    }

    /**
     * Sends given message to all clients in the client list.
     * Actually the message is added to the client sender
     * thread's message queue and this client sender thread
     * is notified to process it.
     */
    private void sendMessageToAllClients(
            String aMessage) {
        for (int i=0; i<mClients.size(); i++) {
            Client client = (Client) mClients.get(i);
            client.mClientSender.sendMessage(aMessage);
        }
    }

    /**
     * Infinitely reads messages from the queue and dispatches
     * them to all clients connected to the server.
     */
    public void run() {
        try {
```

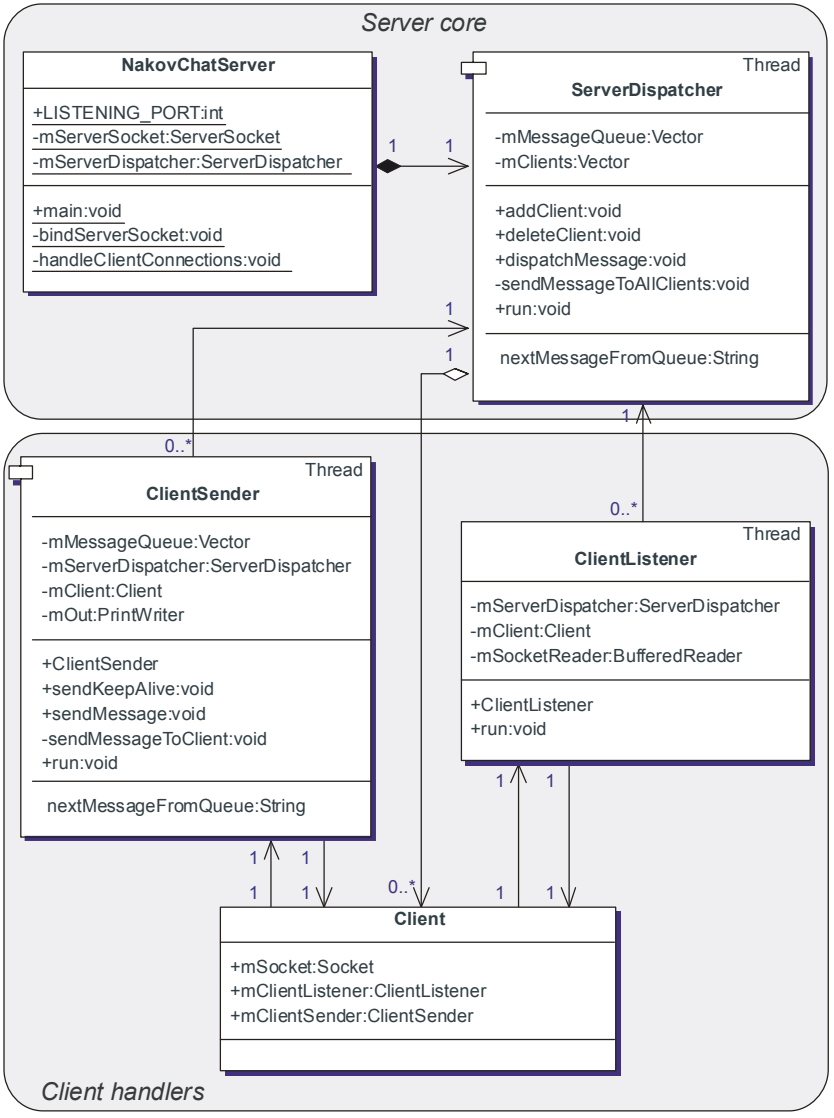
```
            while (true) {
                String message = getNextMessageFromQueue();
                sendMessageToAllClients(message);
            }
        } catch (InterruptedException ie) {
            // Thread interrupted. Stop its execution
        }
    }
}


/**
 * Client class contains information about a client,
 * connected to the server.
 */
class Client {
    public Socket mSocket = null;
    public ClientListener mClientListener = null;
    public ClientSender mClientSender = null;
}


/**
 * ClientListener class listens for client messages and
 * forwards them to ServerDispatcher.
 */
class ClientListener extends Thread {
    private ServerDispatcher mServerDispatcher;
    private Client mClient;
    private BufferedReader mSocketReader;

    public ClientListener(Client aClient, ServerDispatcher
            aSrvDispatcher) throws IOException {
        mClient = aClient;
        mServerDispatcher = aSrvDispatcher;
        Socket socket = aClient.mSocket;
        socket.setSoTimeout(
            NakovChatServer.CLIENT_READ_TIMEOUT);
        mSocketReader = new BufferedReader(
            new InputStreamReader(socket.getInputStream()) );
    }

    /**
     * Until interrupted, reads messages from the client
     * socket, forwards them to the server dispatcher's
     * queue and notifies the server dispatcher.
     */
    public void run() {
        try {
            while (!isInterrupted()) {
                try {
                    String message = mSocketReader.readLine();
```

```
                    if (message == null)
                        break;
                    mServerDispatcher.dispatchMessage(
                        mClient, message);
                } catch (SocketTimeoutException ste) {
                    mClient.mClientSender.sendKeepAlive();
                }
            }
        } catch (IOException ioex) {
            // Problem reading from socket (broken connection)
        }

        // Communication is broken. Interrupt both listener and
        // sender threads
        mClient.mClientSender.interrupt();
        mServerDispatcher.deleteClient(mClient);
    }
}

/**
 * Sends messages to the client. Messages waiting to be sent
 * are stored in a message queue. When the queue is empty,
 * ClientSender falls in sleep until a new message is arrived
 * in the queue. When the queue is not empty, ClientSender
 * sends the messages from the queue to the client socket.
 */
class ClientSender extends Thread {
    private Vector mMessageQueue = new Vector();

    private ServerDispatcher mServerDispatcher;
    private Client mClient;
    private PrintWriter mOut;

    public ClientSender(Client aClient, ServerDispatcher
            aServerDispatcher) throws IOException {
        mClient = aClient;
        mServerDispatcher = aServerDispatcher;
        Socket socket = aClient.mSocket;
        mOut = new PrintWriter(
            new OutputStreamWriter(socket.getOutputStream()) );
    }

    /**
     * Adds given message to the message queue and notifies
     * this thread (actually getNextMessageFromQueue method)
     * that a message is arrived. sendMessage is always called
     * by other threads (ServerDispatcher).
     */
    public synchronized void sendMessage(String aMessage) {
        mMessageQueue.add(aMessage);
        notify();
    }
```

```
    /**
     * Sends a keep-alive message to the client to check if
     * it is still alive. This method is called when the client
     * is inactive too long to prevent serving dead clients.
     */
    public void sendKeepAlive() {
        sendMessage(NakovChatServer.KEEP_ALIVE_MESSAGE);
    }

    /**
     * @return and deletes the next message from the message
     * queue. If the queue is empty, falls in sleep until
     * notified for message arrival by sendMessage method.
     */
    private synchronized String getNextMessageFromQueue()
            throws InterruptedException {
        while (mMessageQueue.size()==0)
            wait();
        String message = (String) mMessageQueue.get(0);
        mMessageQueue.removeElementAt(0);
        return message;
    }

    /**
     * Sends given message to the client's socket.
     */
    private void sendMessageToClient(String aMessage) {
        mOut.println(aMessage);
        mOut.flush();
    }

    /**
     * Until interrupted, reads messages from the message queue
     * and sends them to the client's socket.
     */
    public void run() {
        try {
            while (!isInterrupted()) {
                String message = getNextMessageFromQueue();
                sendMessageToClient(message);
            }
        } catch (Exception e) {
            // Commuication problem
        }

        // Communication is broken. Interrupt both listener
        // and sender threads
        mClient.mClientListener.interrupt();
        mServerDispatcher.deleteClient(mClient);
    }
}
```

Как работи истинският многопотребителски chat сървър

Примерът по-горе се състои от няколко класа, показани на диаграмата:



При стартиране на програмата главният клас на chat сървъра **`NakovChatServer`** създава една нишка **`ServerDispatcher`**, стартира я,

отваря един сървърски TCP сокет и започва да слуша на него постоянно за нови клиенти.

При пристигане на нов клиент **NakovChatServer** създава за него един обект от класа **Client**, и две нишки – **ClientListener** и **ClientSender** съответно за получаване и изпращане на съобщения към този клиент. В **Client** обекта **NakovChatServer** записва сокета на клиента, както и двете нишки, които го обслужват и добавя този обект към списъка с клиентите на нишката **ServerDispatcher**. С това добавянето на нов клиент приключва.

Задачата на нишката **ClientListener** е постоянно да слуша за съобщения идващи от клиента, за който е създадена и при получаване на съобщение, да го изпраща към на **ServerDispatcher** нишката, която има грижата да го достави до всички клиенти. Нишката **ClientListener** прекарва основната част от времето си заспала очаквайки да прочете данни от клиентския сокет.

За четенето се задава timeout от 5 минути. Целта е да се следи за недостъпни клиенти. Ако за 5 минути от клиентския сокет не дойдат никакви данни, клиентът се проверява дали е достъпен (дали има активна връзка до него) като му се изпраща специално служебно keep-alive съобщение. Това се прави, защото ако по даден сокет няма трафик, няма как да се установи дали връзката не се е разпаднала. Така ако няма трафик, идващ от някой клиент в продължение на 5 минути, на този клиент се изпраща keep-alive съобщение. Съответно ако изпращането не успее, ще се установи, че клиентът е недостъпен и връзката с него ще се прекрати. Клиентът трябва да се грижи да игнорира такива съобщения при получаването им.

Нишката **ClientSender** служи да разпраща съобщения до даден клиент, за когото е създадена. През цялото време, когато не разпраща съобщения, тя спи в очакване в опашката й да бъде получено ново съобщение за изпращане към нейния клиент. Нишката **ClientSender** използва модела „производител-потребител” при достъпа до собствената си опашка.

Нишката **ServerDispatcher** служи да разпраща всички подадени й съобщения до всички клиенти, свързани към сървъра. Тя е реализирана с една опашка, в която натрупва всички получени съобщения, които все още не са разпратени. Когато опашката не е празна нишката разпраща съобщенията към диспечерите на всички клиенти, а когато опашката е празна, нишката заспива и чака събуждане. Отново се използва моделът „производител-потребител” при достъпа до опашката.

Във всеки един момент **ServerDispatcher** нишката поддържа списък от всички активни клиенти и следи да актуализирва списъка винаги, когато се прекъсне връзката с някой от клиентите.

Проблемът от предходната реализация на chat сървъра вече е решен ефективно. За всеки клиент в сървъра има отделени специално за него две нишки – една за получаване на съобщения и една за изпращане на съобщения, които работят само за него и спят, когато нямат работа. Всеки клиент се обслужва отделно, сякаш е сам на сървъра и същевременно благодарение на главния диспечер всяко получено съобщение се доставя до опашките за изпращане на всеки от клиентите.

Ако разгледаме внимателно сорс-кода, можем да забележим, че няма особена нужда от **ServerDispatcher** нишката. Единственото, което тя прави, е да поддържа списък от активните клиенти и когато получи съобщение, да го разпрати към техните диспечери. Това наистина е така, защото се очаква операцията изпращане на съобщение към диспечера на даден клиент да не е блокираща операция и да завършва веднага. Единствения смисъл от отделна нишка за разпращането на получените от клиентите съобщения е, че тази нишка позволява клиентът, който е получил съобщението, да си продължи работата веднага вместо да си губи времето да го до опашките на всички свързани клиенти. По този начин би могло леко се подобри скоростта на обслужване на клиента.

Клиент за нашия chat сървър

Нека сега се спрем на една по-проста задача – да създадем клиент за нашия сървър. Клиентското приложение, въпреки че обслужва само един клиент, също трябва да е многонишково, защото обслужването на един клиент реално включва два процеса – получаване на съобщения от сървъра, изпращане на съобщения от клиента към сървъра. И двата процеса в основната част от времето си спят в очакване да получат данни, които да обработят – единият блокира в очакване потребителят да въведе нещо, а другият блокира в очакване от сървъра да се получи нещо. Ето една примерна реализация на chat клиент:

NakovChatClient.java

```
/**
 * Nakov Chat Client
 * (c) Svetlin Nakov, 2002
 * http://www.nakov.com
 */
import java.io.*;
```

```
import java.net.*;

/**
 * NakovChatClient is a client for Nakov Chat Server. After
 * creating a socket connection to the chat server it starts
 * two threads. The first one listens for data comming from
 * the socket and transmits it to the console and the second
 * one listens for data comming from the console and transmits
 * it to the socket. After creating the two threads the main
 * program's thread finishes its execution, but the two data
 * transmitting threads stay running as long as the socket
 * connection is not closed. When the socket connection is
 * closed, the thread that reads it terminates the program
 * execution. Keep-alive messages are ignored when received.
 */
public class NakovChatClient {
    public static final String SERVER_HOSTNAME = "localhost";
    public static String KEEP_ALIVE_MESSAGE = "!keep-alive";
    public static final int SERVER_PORT = 2002;

    private static BufferedReader mSocketReader;
    private static PrintWriter mSocketWriter;

    public static void main(String[] args) {
        // Connect to the chat server
        try {
            Socket socket =
                new Socket(SERVER_HOSTNAME, SERVER_PORT);
            mSocketReader = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
            mSocketWriter = new PrintWriter(new
                OutputStreamWriter(socket.getOutputStream()));
            System.out.println("Connected to server " +
                    SERVER_HOSTNAME + ":" + SERVER_PORT);
        } catch (IOException ioe) {
            System.err.println("Can not connect to " +
                SERVER_HOSTNAME + ":" + SERVER_PORT);
            ioe.printStackTrace();
            System.exit(-1);
        }

        // Start socket --> console transmitter thread
        PrintWriter consoleWriter = new PrintWriter(System.out);
        TextDataTransmitter socketToConsoleTransmitter = new
            TextDataTransmitter(mSocketReader, consoleWriter);
        socketToConsoleTransmitter.setDaemon(false);
        socketToConsoleTransmitter.start();

        // Start console --> socket transmitter thread
        BufferedReader consoleReader = new BufferedReader(
            new InputStreamReader(System.in));
        TextDataTransmitter consoleToSocketTransmitter = new
```

```
            TextDataTransmitter(consoleReader, mSocketWriter);
        consoleToSocketTransmitter.setDaemon(false);
        consoleToSocketTransmitter.start();
    }
}

/**
 * Transmits text data from the given reader to given writer
 * and runs as a separete thread.
 */
class TextDataTransmitter extends Thread {
    private BufferedReader mReader;
    private PrintWriter mWriter;

    public TextDataTransmitter(BufferedReader aReader,
            PrintWriter aWriter) {
        mReader = aReader;
        mWriter = aWriter;
    }

    /**
     * Until interrupted reads a text line from the reader
     * and sends it to the writer.
     */
    public void run() {
        try {
            while (!isInterrupted()) {
                String data = mReader.readLine();
                if (! data.equals(NakovChatClient.
                        KEEP_ALIVE_MESSAGE)) {
                    mWriter.println(data);
                    mWriter.flush();
                }
            }
        } catch (IOException ioe) {
            System.err.println("Lost connection to server.");
            System.exit(-1);
        }
    }
}
```

Основната нишка **NakovChatClient** отваря сокет връзка към chat сървъра, създава 2 нишки, които да обслужват приемането и изпращането на съобщения и след това завършва изпълнението си. Едната от създадените нишки чете постоянно идващите от сървъра съобщения и ги печата на стандартния изход (на конзолата), а другата нишка чете постоянно идващите от стандартния вход (въведените от клавиатурата) съобщения и ги изпраща към сървъра. Ако се прочете служебното съобщение keep-alive, то се игнорира. Ако по време на работа възникне входно-изходен проблем в някоя от двете нишки, това

най-вероятно означава, че се е прекъснала връзката със сървъра и програмата завършва аварийно.

1.5. UDP сокети

В предходната тема изяснихме как се разработват Java приложения, които си комуникират чрез TCP сокети. В тази тема ще се занимаем със средствата, които платформата Java ни дава за комуникация чрез единични UDP пакети.

Предимствата на протокола UDP

Както знаем, протоколът UDP осигурява изпращане и получаване на единични пакети с данни, пристигането на които не е гарантирано. Поради факта, че не установява надеждна връзка между двете приложения, които комуникират по между си, UDP генерира много по-малък мрежов трафик отколкото TCP и затова по принцип осигурява по-голяма бързина при обмяна на единични съобщения.

Кога да използваме UDP

UDP може да се използва, когато трябва да се изпращат малки по размер и независими едно от друго съобщения. Когато се изпращат обемисти съобщения или ако редът на доставянето на съобщенията е важен, UDP не е подходящ избор.

Протоколът UDP има и още една характерна особеност – той е ненадежден. Успешното изпращане на един UDP пакет не гарантира че той ще пристигне или пък че ако изпратим два UDP пакета един след друг, те ще пристигнат в същия ред, в който са изпратени. Ето защо преди да се вземе решение дали да се използва комуникация по UDP, трябва внимателно да се прецени дали този протокол е подходящ.

Къде се използва UDP в практиката

Типичен пример за използване на UDP протокола е при Интернет услугата DNS. При нея клиентът праща кратка заявка, в която описва за кое име на машина или за кой IP адрес иска информация и каква точно информация (такава заявка се нарича DNS query), а DNS сървърът връща кратък отговор (DNS response) с поисканата информация във вид на единичен UDP пакет.

Забележка: Услугата DNS може да работи и по протокол TCP. Вариантът по TCP е по удобен, ако се прави серия от заявки и се търси гарантираност на отговора, докато за единични заявки е по-удобно и по-бързо да се ползва UDP.

Използване на UDP сокети в Java

В Java за поддръжката на UDP сокети разполагаме с класовете **java.net.DatagramSocket** и **java.net.DatagramPacket**. Класът **DatagramSocket** ни дава възможност да се свързваме (да се bind-ваме) към определен мрежов интерфейс и порт и да изпращаме и получаваме пакети. Класът **DatagramPacket** реално представлява структура от данни, която описва един UDP пакет.

Пример за UDP сървър

Да илюстрираме използването на посочените класове чрез един пример. Да си поставим за задача изготвянето на приложение, което при поискване връща на потребителя текущата дата. За получаването на клиентски заявки и изпращането на отговор ще използваме единични UDP пакети. Ето едно възможно решение на поставената задача:

UDPDateServer.java

```
import java.net.*;
import java.util.Date;

public class UDPDateServer {
    public static final int LISTENING_UDP_PORT = 12345;
    public static final String DATE_REQUEST = "GET DATE";
    public static final int RECEIVE_BUFFER_SIZE = 256;

    public static void main(String[] args) throws Exception {
        // Create UDP socket
        DatagramSocket datagramSocket =
            new DatagramSocket(LISTENING_UDP_PORT);
        System.out.println("UDP Date Server is listening " +
            "on port " + LISTENING_UDP_PORT);

        while (true) {
            // Receive UDP client request
            byte[] receiveBuf = new byte[RECEIVE_BUFFER_SIZE];
            DatagramPacket packetIn = new
                DatagramPacket(receiveBuf, receiveBuf.length);
            datagramSocket.receive(packetIn);
            String request =
                new String(receiveBuf, 0, packetIn.getLength());

            // Send response to the client
            if (request.equalsIgnoreCase(DATE_REQUEST)) {
                String response = new Date().toString();
                byte[] responseBuf = response.getBytes();
                InetAddress senderIP = packetIn.getAddress();
                int senderPort = packetIn.getPort();
                DatagramPacket packetOut = new DatagramPacket(
                    responseBuf, responseBuf.length,
```

```
                    senderIP, senderPort);
                datagramSocket.send(packetOut);
            }
        }
    }
}
```

Както се вижда от кода, сървърът отваря един UDP сокет на порт 12345, който се използва както за получаване, така и за изпращане на UDP пакети. След това в безкраен цикъл получава UDP пакет с клиентска заявка (като счита че тя не надвишава 256 байта), извлича от получения пакет IP адреса и порта на изпращача, проверява дали заявката е за извличане на текущата дата, след което създава пакет с отговор (символен низ, съдържащ текущата дата и час) и го изпраща на клиента. Сървърът приема, че клиентът очаква отговора на същия UDP порт, от който е изпратил заявката си. Единствената валидна заявка, на която сървърът отговаря, е за извличане на текущата дата и час. Тази заявка се разпознава по съдържанието на получения UDP пакет, което трябва да е текста „**`GET DATE`**".

Пример за UDP клиент

Нека сега се опитаме да напишем клиент за нашия сървър. Той трябва да изпрати на сървъра UDP пакет, съдържащ заявка за връщане на текущата дата и да слуша известно време за UDP пакет-отговор от сървъра на същия порт, от който е изпратена заявката. Получаването на отговор от сървъра, разбира се, не е гарантирано и затова чакането му трябва да продължи не повече от някакъв предварително зададен времеви лимит (timeout). Ето и примерна реализация:

UDPDateClient.java

```
import java.net.*;

public class UDPDateClient {
    public static final String DATE_SERVER = "localhost";
    public static final int DATE_PORT = 12345;
    public static final String DATE_REQUEST = "GET DATE";
    public static final int TIMEOUT_IN_SECONDS = 5;

    public static void main(String[] args) throws Exception {
        // Send request to the UDP Date Server
        DatagramSocket datagramSocket = new DatagramSocket();
        String request = DATE_REQUEST;
        byte[] requestBuf = request.getBytes();
        DatagramPacket packetOut = new DatagramPacket(
            requestBuf, requestBuf.length,
```

```
            InetAddress.getByName(DATE_SERVER), DATE_PORT);
        datagramSocket.send(packetOut);
        System.out.println("Sent date request to the server.");

        // Receive the server response
        byte[] responseBuf = new byte[256];
        DatagramPacket packetIn =
            new DatagramPacket(responseBuf, responseBuf.length);
        datagramSocket.setSoTimeout(TIMEOUT_IN_SECONDS * 1000);
        try {
            datagramSocket.receive(packetIn);
            String response = new String(
                responseBuf, 0, packetIn.getLength());
            System.out.println("Server response: " + response);
        } catch (SocketTimeoutException ste) {
            System.err.println("Timeout! No response received" +
                " in " + TIMEOUT_IN_SECONDS + " seconds.");
        }
        datagramSocket.close();
    }
}
```

Както се вижда от кода, няма нищо сложно. Първо се създава UDP сокет, след това се изпраща UDP пакет със заявка към сървъра и след това се прави опит в рамките на 5 секунди да се получи отговор. Ако се поучи отговор, той се отпечатва, а иначе се отпечатва съобщение за грешка. Грешката означава, че или сървърът не е получил пакета, или го е получил и е пратил отговор, но отговорът се е изгубил. При протокола UDP изгубването на пакет съвсем не е изключено.

1.6. Multicast сокети

Понякога се налага един пакет да бъде изпратен едновременно до много получатели. Това е необходимо винаги, когато дадено сървърско приложение иска да извести за нещо всички свои клиенти или иска да изпрати една и съща информация едновременно на много потребители. Да вземем за пример излъчването на мултимедийна информация, да кажем цифрова телевизия, в Интернет или локална мрежа. В този случай един и същ мултимедиен сигнал (който на практика е кодиран в някаква последователност от пакети) трябва да достигне едновременно до много потребители. Би било крайно неефективно ако сървърът разпраща информацията на всеки потребител поотделно, особено ако тази информация е обемиста, какъвто е случаят с изпращането на телевизионен сигнал. Например ако в една организация 100 души едновременно гледат един и същ телевизионен канал, не е редно една и съща информация от телевизионния сървър да идва до организацията 100 пъти, като може да дойде само веднъж. Ако се замислим, така е и в реалния свят на спътниковата телевизията. Спътниците излъчват телевизионния сигнал за даден канал само веднъж, а всички потребители, които се интересуват от него го визуализират на своите телевизори. По Интернет не може да стане точно така, но е възможно ако няколко потребителя, които имат общ Интернет доставчик, гледат един и същ канал, информацията от сървъра за телевизия до техния Интернет доставчик да идва само веднъж.

За решаването на описаните проблеми в пакета протоколи TCP/IP има стандартно предвидена функционалност, която позволява групово изпращане на пакети до множество машини. Различават се два модела за групово разпращане на пакети – broadcast и multicast. При broadcast модела един изпратен пакет се получава от всички машини в локалната мрежа, а при multicast модела един пакет се разпраща до всички машини, които предварително са заявили, че искат да го получават, т.е. за се абонирали за дадена група пакети. За разлика от broadcast, multicast може да се използва не само в локална мрежа.

Пример за multicast комуникация

Да си представим, че трябва да изградим комуникационен софтуер за нуждите на голяма организация. Служителите в нея са разделени на работни групи и всеки служител трябва 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 за организацията може да се изгради по следния начин: Софтуерът на всеки служител отваря TCP сокет до централен сървър, през който минават всички съобщения. Сървърът знае към кои групи принадлежи всеки служител 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 и спиране на работата на всички също не е малък.

Multicast сокети в Java

За решаването на проблеми, подобни на описаните по-горе, са разработени multicast сокетите. Те много приличат на UDP сокети, но не са съвсем като тях. Multicast сокетите поддържат 4 основни операции – включване в група (абониране), изпращане на съобщение до дадена група, получаване на съобщение, предназначено за дадена група и изключване от група (спиране на абонамента). Всяка multicast група се идентифицира с уникален IP адрес от диапазона [224.0.0.0 ... 239.255.255.255]. Един компютър може да е едновременно в много multicast групи. Изпращането на пакет до всички членове на дадена група става като се изпрати този пакет до IP адреса на групата.

За регистрация на multicast сокети, както и за включване и изключване в групи се използва класа **`java.net.MulticastSocket`**. При изпращане и получаване на пакети се използва и класа **`java.net.DatagramPacket`** – същият този клас, който се използва за представяне на пакета и при UDP комуникация.

Примерен multicast клиент

Да разгледаме един пример – клиентско приложение, което се абонира за multicast групата 224.0.0.1, след което получава и отпечатва на конзолата всички съобщения, получени на порт 2004 в тази група:

MulticastListener.java

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

```
import java.io.IOException;

public class MulticastListener {
    public static final String
        MULTICAST_GROUP_ADDRESS = "224.0.0.1";
    public static final int LISTENING_PORT = 2004;

    public static final int MAX_PACKET_SIZE = 1024;

    public static void main(String[] args) throws IOException {
        MulticastSocket multicastSocket =
            new MulticastSocket(LISTENING_PORT);
        InetAddress multicastGroupAddr =
            InetAddress.getByName(MULTICAST_GROUP_ADDRESS);
        multicastSocket.joinGroup(multicastGroupAddr);
        System.out.println("Joined to multicast group " +
            MULTICAST_GROUP_ADDRESS + ".");

        byte[] receiveBuf = new byte[MAX_PACKET_SIZE];
        DatagramPacket packet =
            new DatagramPacket(receiveBuf, receiveBuf.length);
        System.out.println("Listening for packets...");
        while (true) {
            multicastSocket.receive(packet);
            String packetAsString = new String(
                packet.getData(), 0, packet.getLength());
            System.out.println("Received packet from " +
                packet.getAddress().getHostAddress() + ":" +
                packet.getPort() + " - " + packetAsString);
        }
    }
}
```

Както виждаме от кода, работата с multicast сокети съвсем не е сложна. Създаваме си 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 на избрания порт.

Използвахме класа **`java.net.MulticastSocket`**, който предоставя няколко основни метода – **`joinGroup()`** за присъединяване към multicast група, **`leaveGroup()`** за напускане на група, **`getTTL()`** и **`setTTL()`** за извличане и промяна на параметъра TTL (time to live).

Как работи multicasting-а. Какво е IGMP. Какво е TTL

Преди да си изясним каква е ролята на TTL параметъра, трябва да си изясним механизма, по който работи multicast комуникацията. За нея от съществено значение е протоколът IGMP (Internet Group Management Protocol). IGMP е протокол на мрежово ниво, част от комплекта

протоколи TCP/IP и служи да информира маршрутизаторите в дадена мрежа за това, че даден хост иска или не иска да получава съобщенията за дадена multicast група. По принцип пакетите, предназначени за даден multicast адрес могат да преминават от един маршрутизатор към друг и да достигат съседни мрежи, т.е. multicast комуникацията може да работи не само в локална мрежа, но и в Интернет. Преминаването на един пакет през един маршрутизатор намалява стойността на неговия 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.

Изпращането и получаването на multicast пакети става по протокол UDP. Няма разлика между това дали изпращаме обикновен UDP пакет и UDP пакет до multicast адрес. Разликата е само в адреса на получателя. Ако адресът е някаква multicast група, пакетът ще бъде доставен до всички нейни членове, а ако адресът е обикновено IP, пакетът ще бъде доставен само до съответния хост. Абонирането и прекратяването на абонамент към дадена група става по протокол IGMP, но след успешен абонамент вече се използва обикновена UDP комуникация.

Каква е ефективността при multicast комуникация

Поради факта, че за разпространението на multicast UDP пакетите се грижи мрежовият хардуер на IP ниво, този метод за разпращане на съобщения до група потребители е изключително ефективен и многократно по-бърз от алтернативните подходи с централен сървър и TCP или UDP базирана комуникация. Ефективността може да се разгледа от две различни страни:

- спестява се време при изпращане на едно и също съобщение до голям брой потребители
- спестява се мрежов трафик

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 сокет или обикновен UDP пакет.

От друга страна, когато едно съобщение трябва да се получи от хиляди машини в дадена мрежа, то достига до тази мрежа само веднъж и всички машини го получават без това да причини повече трафик отколкото ако само една машина в тази мрежа трябва да го получи. Така мрежовият трафик може да се намали хиляди пъти.

Заради голямата си ефективност при комуникация в група multicasting-ът е предпочитан метод за работа при приложения, за които високата производителност е от жизнена важност.

Примерен multicast сървър

След като изяснихме в общи линии какво е multicasting и за какво се използва, нека напишем сървър за нашия примерен multicast клиент. Да си поставим за задача създаването на програмка, която изпраща на всяка секунда по едно съобщение до всички машини в multicast групата 224.0.0.1, като го адресира до порт 2004, на който би могъл да слуша нашият примерен multicast клиент. Ето една примерна реализация:

MulticastSender.java

```
import java.net.*;

public class MulticastSender {
    public static final String
        MULTICAST_GROUP_ADDRESS = "224.0.0.1";
    public static final int TARGET_PORT = 2004;

    public static void main(String[] args) throws Exception {
        InetAddress multicastGroupAddr =
            InetAddress.getByName(MULTICAST_GROUP_ADDRESS);
        MulticastSocket multicastSocket = new MulticastSocket();
        while (true) {
            String message = "Hello " + new java.util.Date();
            DatagramPacket packet = new DatagramPacket(
                message.getBytes(), message.length(),
                multicastGroupAddr, TARGET_PORT);
            multicastSocket.send(packet);
            System.out.println("Sent UDP packet to " +
                MULTICAST_GROUP_ADDRESS + ":" + TARGET_PORT);
            Thread.sleep(1000);
        }
    }
}
```

Както виждаме, кодът силно прилича на код, който разпраща UDP пакети. Единствената разлика е, че вместо класа **`DatagramSocket`** е използван класът **`MulticastSocket`** и преди да започне разпращането се указва групата, към която ще става изпращането. Примерната сървърска програма в безкраен цикъл праща съобщение, съдържащо текущата дата.

Как работят примерния multicast клиент и сървър

За да тестваме клиента и сървъра, компютърът ни трябва да е включен в мрежа. На компютър без мрежов интерфейс и двете програми биха могли да не работят (всъщност това зависи до известна степен от операционната система). Най-добре можем да видим как работят примерния клиент и примерния сървър ако имаме няколко компютъра в мрежа и пуснем на всеки от тях няколко клиента и няколко сървъра. 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. Ето как биха могли да изглеждат съответно изходите на един от клиентите и на един от сървърите ако в мрежата има пуснати 2 клиента и 2 сървъра съответно на машини с IP адреси 192.168.200.1 и 192.168.200.2 :

Изход от MulticastListener на машината 192.168.200.1

```
Joined to multicast group 224.0.0.1.
Listening for packets...
Received packet from 192.168.200.2:1448 - Hello Sat Mar 06
23:35:34 EET 2004
Received packet from 192.168.200.1:4107 - Hello Sat Mar 06
23:35:46 EET 2004
Received packet from 192.168.200.2:1448 - Hello Sat Mar 06
23:35:35 EET 2004
Received packet from 192.168.200.1:4107 - Hello Sat Mar 06
23:35:47 EET 2004
Received packet from 192.168.200.2:1448 - Hello Sat Mar 06
23:35:36 EET 2004
...
```

Изходът на сървърът не е толкова интересен:

Изход от MulticastSender на машината 192.168.200.1

```
Sent UDP packet to 224.0.0.1:2004
Sent UDP packet to 224.0.0.1:2004
Sent UDP packet to 224.0.0.1:2004
Sent UDP packet to 224.0.0.1:2004
...
```

1.7. Работа с URL ресурси

Java е език за програмиране от високо ниво, който още от самото си създаване е бил силно ориентиран към работа с Интернет. Извличането на ресурси от WWW с Java може да бъде изключително лесно, ако използваме класа **`java.net.URL`**, но преди да видим как става това, нека си изясним какво е това URL.

Какво е URL

Както при пощенските услуги, за доставяне на някаква поща е необходим адреса на получателя (държава, град, улица, номер и т.н.), така и в Интернет, в рамките на глобалната разпределена информационна система World Wide Web (WWW), за достъп до някакъв ресурс е необходим неговият адрес. Адресите на ресурси 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. Може да бъде `http`, `ftp`, `https` 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 незадължително поле, което указва номера на порта на машината, зададена в полето host, примерно **`80`** или **`8080`**. Ако номер на порт не е зададен, се използва портът по подразбиране за дадения протокол, например 80 за HTTP, 21 за FTP, 443 за HTTPS и т.н.

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 URL-ът на ресурса, който искаме да извлечем, е http://www.nakov.com/about/CV-Svetlin-Nakov.html, протоколът е **`http`**, хостът е `www.nakov.com`, портът не е зададен и се подразбира че е стандартния за http – **`80`**, а ресурсът е **`/about/CV-Svetlin-Nakov.html`**, като **`/about/`** е пътят до ресурса, а **`CV-Svetlin-Nakov.html`** е името му. Пътят **`/about/`** до ресурса се нарича още виртуална директория на Web-сървъра.

Как можем да извлечем в Java ресурс по даден URL

Всички ние всекидневно използваме URL адреси за достъп до различни сайтове докато си сърфираме из Интернет. Когато използваме стандартен Web-браузър, обикновено пишем в полето за адрес URL

адреса на сайта, който искаме да посетим и натискаме бутона за извличане на зададения адрес.

Когато работим с Java нещата са подобни – отваряме URL връзка като създаваме обект от класа **java.net.URL**, задаваме адреса на ресурса, от който се интересуваме и го извличаме използвайки стандартен входен поток. Да илюстрираме това с един малък пример – програма, която извлича документа http://www.nakov.com/about/CV-Svetlin-Nakov.html и го отпечатва на стандартния изход:

RetrieveURLExample.java

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

public class RetrieveURLExample {
    public static void main(String[] args) throws IOException {
        URL url = new URL(
            "http://www.nakov.com/about/CV-Svetlin-Nakov.html");
        BufferedReader in = new BufferedReader(
            new InputStreamReader(url.openStream()));
        String line;
        while ((line = in.readLine()) != null)
            System.out.println(line);
        in.close();
    }
}
```

Кодът е съвсем кратък и ясен – създаваме обект от класа **URL**, като му подаваме адреса на ресурса, до който искаме да установим достъп, след което отваряме входен поток за четене на този ресурс с метода **openStream()** и от този поток прочитаме целия ресурс ред по ред. В случая 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.

Писане в URL ресурс

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 може да се направи това:

AltaVistaSearch.java

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

```
public class AltaVistaSearch {
    public static void main(String[] args)
    throws Exception {
        URL url = new URL("http://www.altavista.com/search");

        // Send request
        URLConnection connection = url.openConnection();
        connection.setDoOutput(true);
        PrintWriter out = new PrintWriter(
            connection.getOutputStream());
        String query =
            URLEncoder.encode("Svetlin Nakov", "UTF-8");
        out.println("q=" + query);
        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 търсачката AltaVista, като й се задава да търси фразата „Svetlin Nakov". Понеже търсачката работи по протокол HTTP, е необходимо всичко, което й се изпраща, да се кодира съгласно HTTP стандарта, така че да бъде разпознато коректно от Web сървъра. За целта се използва класа URLEncoder, с който параметрите се кодират по стандарта за URL. След това се отваря изходен поток към ресурса, през който на заявката й се подават кодираните параметри и накрая се отваря входен поток за четене от ресурса за да се прочете резултатът от извършеното в AltaVista търсене. За писане в URL се използва обект от класа URLConnection, който се взима с метода **`openConnection()`**.

Писането писане в URL ресурс, достъпен по протокол HTTP на практика означава да се изпрати HTTP POST заявка към дадения ресурс, като се дава възможност да се изпратят и параметри. В нашия пример на сървъра **`www.altavista.com`** му се задава HTTP POST заявка, която съдържа адреса на динамичния ресурс **`/search`**, на който се задава един единствен параметър с име **`q`** и стойност **`Svetlin Nakov`**, кодирана в UTF-8.

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» ***Светлин Наков*** е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» ***Мартин Кулов*** е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Глава 2. Java аплети

В следващите няколко теми ще разгледаме технологията на Java аплетите, която ни позволява в HTML документи да вграждаме Java програмен код, който се изпълнява от Web-браузъра на потребителя при разглеждане на HTML документа. Поради факта, че тази технология бавно и лека полека започва да губи позиции за сметка на някои по-нови технологии (като например .NET Windows Forms Controls, Macromedia Flash и SVG контролите), ще си позволим да не й обърнем прекалено голямо внимание.

Първоначално ще се запознаем със същността на аплетите, ще дадем един пример и ще обясним възможностите за вграждане на аплет в HTML документ.

По-нататък ще продължим с изясняване на жизнения цикъл на аплетите.

След това ще направим кратък преглед на средствата, които Java платформата предоставя за създаване на прозоречно-ориентиран графичен потребителски интерфейс като обърнем внимание на някои от най-основни елементи на библиотеката AWT. Няма да си поставяме за цел да разгледаме цялата библиотека AWT нито нейните основни концепции и програмният й модел, защото това са специфични знания, които не са прекалено необходими на един Интернет разработчик. Очаква се читателите или да имат някакви основни познания по AWT или да разполагат със специализиран редактор за AWT-базиран графичен потребителски интерфейс, който да им помогне бързо да навлязат в материята, без да има нужда да я познават добре.

В последната част на темата за аплетите ще обясним какви са ограниченията, които са им наложени от съображения за сигурност. Ще видим как въпреки ограничените права, с които аплетите се изпълняват, можем да ги използваме за асинхронна сокет-базирана комуникация със сървъра, която стандартно не се поддържа във Web среда.

2.1. Въведение в Java аплетите

В първите години след като Java се появи, беше силно разпространено схващането, че този език служи единствено за създаване на аплетчета, които разширяват стандартните възможности на HTML и JavaScript и правят Web-страничките „по-раздвижени". По това време такова схващане беше в голяма степен правилно, защото Java първоначално се използваше точно за това и масовото му използване за по-общи задачи започна по-късно. В следващите няколко години Java платформата постепенно се наложи като основен играч в софтуерната индустрия и в световен мащаб се утвърди като една от най-предпочитаните платформи за разработка на сложни корпоративни приложения.

Какво представляват Java аплетите

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 браузърът им дава, за графичния си потребителски интерфейс. Аплетите притежават почти цялата мощ която ни дава Java платформата, но с известни ограничения, наложени главно от съображения за сигурност. Те представляват компилирана Java програма във вид на .class файл или съвкупност от компилирани Java класове, записани в .jar архив.

Както знаем, всички Java програми се изпълняват от Java виртуална машина (JVM) и затова всички браузъри, които поддържат аплети, имат или вградена в себе си или допълнително инсталирана виртуална машина. При отварянето на HTML документ, съдържащ аплет, браузърът зарежда Java виртуалната си машина и стартира аплета в нея.

За да се осигури безопасността на потребителите аплетите се изпълняват с понижени права. Те нямат достъп до локалната файлова система, не могат да комуникират свободно по мрежата, не могат да изпълняват и да си комуникират с други програми, не могат да изпълняват системни функции, не могат да извличат информация за обкръжаващата ги среда и не могат да правят още много други действия, които застрашават по някакъв начин сигурността на потребителя или безопасността на неговите лични данни.

Как се създават Java аплети

Най-често Java аплетите наследяват класа **java.applet.Applet** и припокриват методите му за инициализация и за изчертаване върху 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 малко по-късно. Да се спрем за начало на един съвсем прост пример за аплет:

HelloWorldApplet.java

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

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

Единственото, което прави аплетът от този пример, 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.

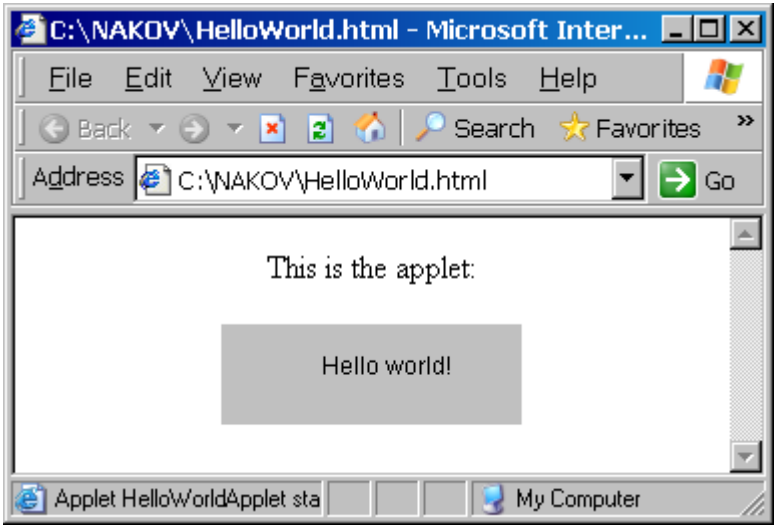
Как се изпълняват Java аплети

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. Една възможност да направим това е следната::

HelloWorld.html

```
<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 **HelloWorld.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 много повече права на аплета. Като параметър appletviewer приема име на HTML файл, който съдържа аплет.

Класът java.applet.Applet

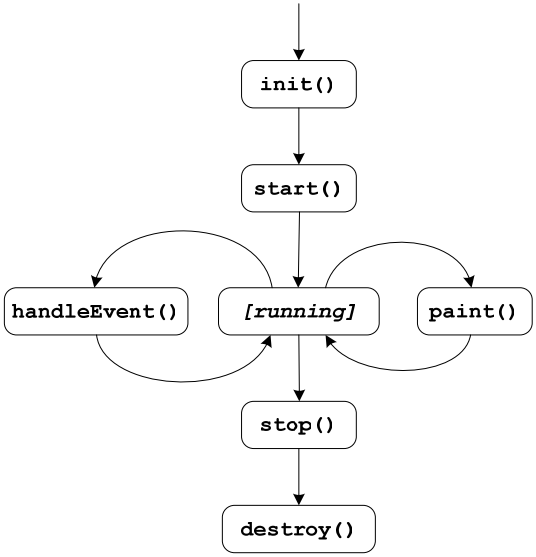
Класът **`java.applet.Applet`** е базов клас за всички Java аплети и е наследник на класа **`java.awt.Panel`**. Той представлява стандартен AWT контейнер и следователно в него могат да се поставят различни AWT компоненти с цел изграждане на потребителски интерфейс. Освен за поставяне на компоненти, пространството на един аплет може да се използва и за директно рисуване със средствата на AWT, както това е направено в горния пример. Директното рисуване във вътрешността на аплета трябва да се прави само от метода **`paint()`**, защото иначе има риск да се наруши целостта на нарисуваното изображение ако то бъде застъпено от някакъв друг прозорец. Методът **`paint()`** се извиква винаги, когато аплетът има нужда да се пречертае.

Жизнен цикъл на аплетите

Класът **`java.applet.Applet`** дава базовата функционалност на аплетите и предоставя на наследниците си методи за взаимодействие с браузъра и външния свят. Нека се опитаме да проследим какво се

случва от момента на зареждане на HTML-страницата с аплета до момента, в който се затвори браузърът или се премине на друга страница. Първоначално браузърът 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.

Нека разгледаме жизнения цикъл на един аплет:



Както се вижда от диаграмата, изпълнението на аплет включва няколко отделни етапа: Първоначално се извиква се **init()** метода, а след него и **start()** метода. Докато аплетът работи, браузърът 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 други такива 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 по подразбиране на тези събития предизвиква извикване на метода **paint()**, при изпълнението на който аплетът е длъжен да се пречертае. При приключване на изпълнението на аплета, браузърът 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 и повече от веднъж. Методът **`stop()`** се извиква, когато браузърът 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()`**. Поради липсата на точна спецификация различните Web-браузъри реализират по различен начин **`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 браузъра. Може да се използва за освобождаване на ресурси.

Библиотеката AWT

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 взаимодействат активно с потребителя. Този framework ни предоставя стандартен начин за работа с графични контроли, като например прозорци, диалози, менюта, бутони, текстови полета, картинки, текст и др., а също и механизъм за обработка на събитията възникнали в резултат от взаимодействието между потребителя и програмата, като щракване с мишката, вход от клавиатурата и др.

Работата с AWT не е толкова проста и интуитивна, както при някои други компонентно-ориентирани библиотеки за създаване на графичен потребителски интерфейс, като например VCL библиотеката в Delphi и Windows Forms библиотеката в Microsoft .NET. Заради стремежът си да бъде мощна, универсална и лесно преносима библиотека AWT е станала сложна за използване и доста тромава, заради което вина има и самата архитектура на Java виртуалната машина, заради която повечето Java приложение са традиционно няколко пъти по-бавни от аналогични **native** приложения. Наистина едно от най-бавните неща в Java платформата са стандартните й средства за работа с графичен потребителски интерфейс – библиотеките AWT и Swing.

Архитектура Model-View-Controller

В архитектурно отношение AWT налага програмния модел MVC (model-view-controller) – модел, при който има строго разделение между данните (model), контролите, които ги визуализират (view) и логиката, която управлява извършените от потребителя действия (controller). Например една таблица в AWT е визуална контрола (view), която за да работи, изисква клас, който да й подава данните за визуализация (model) и клас, който да управлява извършените от потребителя действие (controller).

Аплетите и AWT

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 във вътрешността на аплет става чрез класа **`java.awt.Graphics`**. Обекта от този клас, който съответства на аплета, се подава автоматично при всяко извикване на **`paint()`** метода му. Всеки **`Graphics`** обект има своя собствена координатна система и всеки AWT графичен компонент има свой собствен **`Graphics`** обект, чрез който той 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. Описания на методите, с които се чертаят тези обекти, като **`drawLine()`**, **`drawRect()`**, **`fillRect()`**, **`clearRect()`**, **`drawOval()`**, **`fillOval()`**, **`drawArc()`**, **`fillArc()`**, **`drawPolygon()`**, **`fillPolygon()`** и др. могат да се намерят в документацията на JDK.

Използване на картинки от аплет

Освен директното чертане на геометрични фигури, AWT позволява и изобразяване на картинки, заредени от GIF или JPEG файлове. 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 използваме следния код:

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

Ако искаме да начертаем картинката с променени размери, можем да използваме същия метод **drawImage()**, но с други параметри:

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

Тънкости при работата с картинки в AWT

Внимателният читател вероятно е забелязал, че методът **drawImage()** приема 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 става 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 методът **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. Най-често реализацията на метода **imageUpdate()** пречертава картинката, т.е. извиква метода **drawImage()**.

Стандартно 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 завършила своето зареждане. Използвайки тази базова функционалност на класа **Applet**, можем винаги, когато зареждаме картинки от аплет, да подаваме за **ImageObserver** самия аплет, т.е. обекта **this**.

Когато зареждаме картинки винаги трябва да внимаваме да не разчитаме, че дадена картинка ще се зареди веднага. Картинките винаги се зареждат асинхронно и затова трябва да се грижим да ги чертаем едва след като са заредени или поне да ги пречертаваме след това.

За изчакване на асинхронното зареждане на картинка в AWT има специален клас **java.awt.MediaTracker**, на който чрез метода **addImage(…)** може да му се подават картинки, които са в процес на зареждане, след което може да се изчака приключването на зареждането на всички картинки чрез метода **waitForAll()**.

Пример за аплет за анимация

Да разгледаме един цялостен пример за аплет, който използва картинки и реализира проста анимация чрез допълнителна нишка. Да си поставим за задача направата на Java аплет, в който една топка постоянно се движи и се отблъсква в стените на аплета при удар (както при игра на билярд). Едно възможно решение на задачата е следното:

BallApplet.java

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

public class BallApplet extends Applet implements Runnable {
    public static final int ANIMATION_SPEED = 10;
    public static final String IMAGE_NAME_PARAM = "imageName";

    private int mBallX, mBallY, mBallSpeedX, mBallSpeedY;
    private Image mBallImage;
    private Image mImageBuffer;
    private Graphics mImageBufferGraphics;

    private Thread mAnimationThread;
    private boolean mAnimationThreadInterrupted = false;

    /**
     * Applet's init() method. Makes some initializations
     * and loads the ball image. This method is called before
     * creating the animation thread so no synchronization
     * is needed.
     */
    public void init() {
        // Load the ball image from the server
        String imageName = getParameter(IMAGE_NAME_PARAM);
        if (imageName == null) {
            System.err.println("Applet parameter " +
                IMAGE_NAME_PARAM + " is missing.");
            System.exit(-1);
        }
        mBallImage = getImage(getCodeBase(), imageName);

        // Wait for the image to load completely
        MediaTracker tracker = new MediaTracker(this);
        tracker.addImage(mBallImage,0);
```

```
        try {
            tracker.waitForAll();
        } catch (InterruptedException ie) { }
        if (tracker.statusAll(true) != MediaTracker.COMPLETE) {
            System.err.println("Can not load " + imageName);
            System.exit(-1);
        }

        // Initialize the ball image coordinates and speed
        mBallX = 1;
        mBallY = 1;
        mBallSpeedX = 1;
        mBallSpeedY = 1;

        // Create an image buffer for the animation
        mImageBuffer = createImage(
            getSize().width, getSize().height);
        mImageBufferGraphics = mImageBuffer.getGraphics();
    }

    /**
     * Applet's paint() method. Draws the ball on its current
     * position. This method can be called in the same time
     * from both the applet's thread and from the animation
     * thread so it should be thread safe (synchronized).
     */
    public void paint(Graphics aGraphics) {
        synchronized (this) {
            if (mAnimationThread != null) {
                // Paint in the buffer
                mImageBufferGraphics.fillRect(
                    0, 0, getSize().width, getSize().height);
                mImageBufferGraphics.drawImage(
                    mBallImage, mBallX, mBallY, this);

                // Copy the buffer contents to the screen
                aGraphics.drawImage(mImageBuffer, 0, 0, this);
            }
        }
    }

    /**
     * Applet's start() method. Creates the animation thread
     * and starts it if it is not already running. This method
     * can be called only from the applet's thread so it does
     * not require synchronization.
     */
    public void start() {
        if (mAnimationThread == null) {
            mAnimationThreadInterrupted = false;
            mAnimationThread = new Thread(this);
            mAnimationThread.start();
```

```
        }
    }

    /**
     * Applet's stop() method. Asks the animation thread to
     * stop its execution and waits until it is really stopped.
     * This method is called only from the applet's thread so
     * it does not need synchronization except when accessing
     * the variable mAnimationThreadInterrupted that is common
     * for applet's thread and animation thread.
     */
    public void stop() {
        synchronized (this) {
            mAnimationThreadInterrupted = true;
        }
        try {
            mAnimationThread.join();
        } catch (InterruptedException ie) { }
        mAnimationThread = null;
    }

    /**
     * Animation thread's run() method. Continuously changes
     * the ball position and redraws it and thus an animation
     * effect is achived. This method runs in a separate thread
     * that is especially created for the animation. A
     * synchronization is needed only when accessing variables
     * that are common for the applet's thread and animation
     * thread.
     */
    public void run() {
        // Calculate the animation area size
        int maxX, maxY;
        synchronized (this) {
            maxX = this.getSize().width -
                mBallImage.getWidth(this);
            maxY = this.getSize().height -
                mBallImage.getHeight(this);
        }

        // Perform continuously animation
        while (true) {
            synchronized (this) {
                // Check if the thread should stop
                if (mAnimationThreadInterrupted)
                    break;

                // Calculate the new ball coordinates
                if ((mBallX >= maxX) || (mBallX <= 0))
                    mBallSpeedX = -mBallSpeedX;
                mBallX = mBallX + mBallSpeedX;
                if ((mBallY >= maxY) || (mBallY <= 0))
```

```
                    mBallSpeedY = -mBallSpeedY;
                mBallY = mBallY + mBallSpeedY;
            }

            // Redraw the applet contents
            paint(getGraphics());

            // Wait some time to slow down the animation speed
            try {
                Thread.sleep(ANIMATION_SPEED);
            } catch (Exception ex) {}
        }
    }
}
```

Как работи аплета за анимация

Създаването на анимация по принцип не е много проста работа, защото изисква управление на нишки. Необходимо е добро познаване на жизнения цикъл на аплетите, познаване на библиотеката AWT, умения за работа с нишки и синхронизация на достъпа до общи ресурси.

Нашият примерен аплет за анимация работи по следния начин: При инициализация, в метода **`init()`**, аплетът зарежда картинката, която ще се движи (в нашия случай това е топка), инициализира координатите и посоката й на движение и създава специален буфер за чертане за целите на анимацията.

При стартиране на аплета, когато се извиква методът му **`start()`**, се създава една нишка, която се грижи за анимацията. Всичко, което тя прави е да променя през определено време (в случая 10 милисекунди) координатите по **`x`** и по **`y`** на топката съгласно текущата посока на движение, да сменя посоката на движение при удар в стена и след всяка промяна на координатите на топката да пречертава цялото съдържание на аплета. Методът **`start()`** няма нужда от синхронизация, защото когато се изпълнява единствената работеща нишка е нишката на аплета.

При извикване на **`stop()`** метода аплетът спира нишката и я изчаква да приключи изпълнението си. Когато този метод се извика е възможно да работят едновременно две нишки – нишката на аплета и нишката за анимация и затова е необходима синхронизация при достъпа до общи за двете нишки променливи.

Пречертаването на аплета (**`paint(…)`** метода) работи с буфериране за да се избегне трепкането, което би се получило ако се изтрие съдържанието на аплета и след това се начертае върху него движещият се обект на текущата му позиция. Пречертаването е операция, която може

да се извика едновременно от две различни нишки. Нишката на аплета може по всяко време да извика пречертаване заради засичане на аплета с друг прозорец или по някаква друга причина, а нишката за анимация също по всяко време да извика пречертаване заради нуждата от движение на топката. За да не се засичат две пречертавания, понеже те биха използвали един и същ работен буфер, е необходимо пречертаването да е синхронизирана операция.

Техниката „двойно буфериране”

Пречертаването на аплета работи със специален буфер. Този буфер се използва за да се избегне премигването и да се получи наистина плавно движение. При всяко пречертаване на аплета буферът се изчиства с **`fillRect()`**, след това в него се начертава топката на текущата й позиция и на екрана се изобразява съдържанието на този буфер. Тази техника за избягване на премигването при създаване на анимация се нарича „двойно буфериране” (double buffering). При стартиране на аплета се създава обект от класа **`Image`** и се работи чрез неговия **`Graphics`** обект. Вместо да се рисува директно върху аплета, се рисува в буфера и след това нарисуваният вече кадър от буфера се прехвърля върху повърхността на аплета.

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, ако се налага ако тези настройки бъдат променени. За задаването им има специален таг, който се влага в така **`<applet>`** – тага **`<param>`**.

Стартиране на аплета за анимация

Ето един примерен HTML код, който стартира нашия аплет и задава за параметъра име на картинка **`imageName`** стойността **`ball.jpg`**:

BallAppletTest.html

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

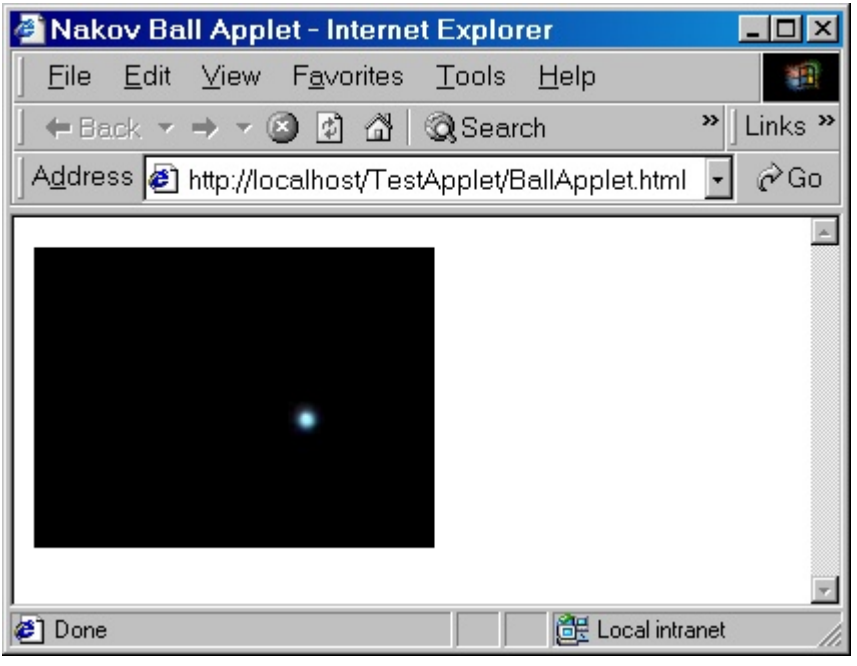
Разбира се, в директорията, където е записан този **`html`** файл е необходимо да запишем компилирания аплет **`BallApplet.class`**, както и файла с топката, която ще подскача в аплета – **`ball.jpg`.**

В инициализационната си част аплетът взима параметъра **`imageName`** и зарежда картинката с това име от същата директория, 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, зададен спрямо директорията, от която е зареден аплетът.

След като е извикан методът за зареждане на картинка тя е започва да се зарежда от зададения URL. Понеже, както знаем, работата с картинки в AWT е асинхронна, е необходимо да изчакаме картинката с топката да се зареди напълно преди да се обръщаме към нея. За целта се използва класът **`MediaTracker`**, чрез който може да се проследи състоянието на започнали да се зареждат картинки.

Аплетът за анимация в действие

Ето как изглежда резултата 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.

2.2. Особености на аплетите и работата с AWT

В настоящата тема ще разгледаме някои особености на аплетите, на начина им на изпълнение, особеностите на различните Web-браузъри и различните виртуални машини, както и някои тънкости свързани с начина на вграждане на аплети в HTML документи.

Ще продължим с един пример за използване на графичните контроли от библиотеката AWT за създаване на прозоречно-ориентиран графичен потребителски интерфейс. Ще си поставим за задача направата на прост аплет-калкулатор, с който ще илюстрираме как се използват текстови полета и бутони и как можем да прихващаме събитията, които се генерират от тях при взаимодействието им с потребителя.

Web-браузърите кешират аплетите

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 подобряване на скоростта на зареждане на документа. Кеширането се използва и за аплетите, поради което трябва да се внимава. Записването на нова версия на аплета в директорията, от който той се зарежда и натискане на бутона “Refresh” не гарантира изпълнението на новата версия. 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 биха могли да са други. Най-сигурния начин да сме сигурни, че тестваме последната версия на разработвания аплет след промяна в кода е като го прекомпилираме, а след това затворим браузъра и го стартиране отново.

Java виртуалната машина при различните Web-браузъри

Друга важна особеност на браузърите е че повечето поддържат стандартно само много стари версии на JDK. Например Internet Explorer 4.0, 5.0 и 5.5 поддържат само JDK 1.1, какъвто е и случаят с повечето версии на Netscape Navigator. JDK 1.3, 1.4 и следващите версии се поддържат само след като се издърпа и инсталира продуктът **Java Plug-In** от сайта на Sun – http://java.sun.com/products/plugin/. Поради лошите отношения между Microsoft и Sun, Inetrnet Explorer 6.0 вече въобще не поддържа стандартно аплети и при него използването на Java Plug-In е задължително. Web-браузърите Netscape 6 и Mozilla 1.x също не подържат стандартно Java аплети и се нуждаят от Java Plug-In.

Поради изтъкнатите проблеми със съвместимостта когато пишем аплети трябва да използваме или JDK версия 1.1 или да изискваме потребителят да има инсталиран Java Plug-In по-висока версия.

Ако решим да използваме JDK 1.1 трябва да знаем, че повечето класове и методи, които биха ни потрябвали при писане на аплети, ги има в JDK 1.1, но имената на някои методи в различните версии на JDK са различни, въпреки че 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`**.

Java конзолата и проследяването на проблеми с аплетите

При използването на липсващ метод от някой клас, виртуалната машина на браузъра предизвиква изключение. Изключения се предизвикват и при много други ситуации, които се срещат и при обикновените Java програми. 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 4.x и 5.x 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 браузърът. Ако се използва Java Plug-In конзолата е достъпна от самия него.

Намирането на проблем в аплет без Java конзолата е почти немислимо, така че когато разработвате аплети и нещо не работи, винаги поглеждайте в нея. Типичен е случаят, в който в средата за разработка (например JBuilder, IDEA или Eclipse) аплетите работят, а в браузъра не тръгват. Причините могат да са две – или има разлика във версиите на JDK в средата за разработка и поддържаната в браузъра или аплетът се опитва да извърши нещо за което няма права.

Обикновено **`appletviewer`** или средата за разработка стартират аплетите с повече права, отколкото един браузър би им дал и с JDK, по-висока версия от 1.1 и затова се получават несъвместимости.

Пример за аплет-калкулатор

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 с 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 такъв аплет:

SumatorApplet.java

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

public class SumatorApplet extends Applet {
    private TextField mNumber1Field = new TextField();
    private TextField mNumber2Field = new TextField();
    private Button mCalcButton = new Button();
    private TextField mSumField = new TextField();
    private Color mLastBackgroundColor;

    public void init() {
        this.setBackground(Color.black);

        // Set layout manager to null
        this.setLayout(null);

        // Create the first text field
        mNumber1Field.setBounds(new Rectangle(20, 50, 60, 25));
        mNumber1Field.setBackground(Color.white);
        this.add(mNumber1Field, null);

        // Create the second text field
        mNumber2Field.setBounds(new Rectangle(95, 50, 60, 25));
        mNumber2Field.setBackground(Color.white);
        this.add(mNumber2Field, null);

        // Create the "calculate sum" button
        mCalcButton.setBounds(new Rectangle(170, 50, 90, 25));
        mCalcButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    calcSum();
                }
            });
        mCalcButton.setLabel("calc sum");
        this.add(mCalcButton, null);

        // Create the result text field
        mSumField.setEditable(false);
        mSumField.setBackground(Color.gray);
        mSumField.setForeground(Color.white);
        mSumField.setBounds(new Rectangle(20, 85, 240, 25));
        this.add(mSumField, null);
```

```
    }

    public boolean mouseDown(Event aEvent, int aX, int aY) {
        mLastBackgroundColor = this.getBackground();
        this.setBackground(Color.red);
        return true;
    }

    public boolean mouseUp(Event aEvent, int aX, int aY) {
        this.setBackground(mLastBackgroundColor);
        return true;
    }

    private void calcSum() {
        try {
            long s1 = new Long(mNumber1Field.
                getText()).longValue();
            long s2 = new Long(mNumber2Field.
                getText()).longValue();
            mSumField.setText(s1 + " + " + s2 + " = " +
                (s1+s2));
        } catch (Exception ex) {
            mSumField.setText("Error!");
        }
    }

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

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

Как работи аплетът-калкулатор

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.

Като следваща стъпка създаваме компонентите на аплета една по една – първото текстово поле, второто текстово поле и накрая бутона.

За да прихванем събитието "натискане 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`** чрез 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 текста 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 създава инстанция на аплета и един обект **`java.awt.Frame`**, в който се той се поставя, 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 браузъра, защото, както знаете аплетите се държат по различен начин в различните среди, в които се стартират.

Изпълнение на аплета-калкулатор

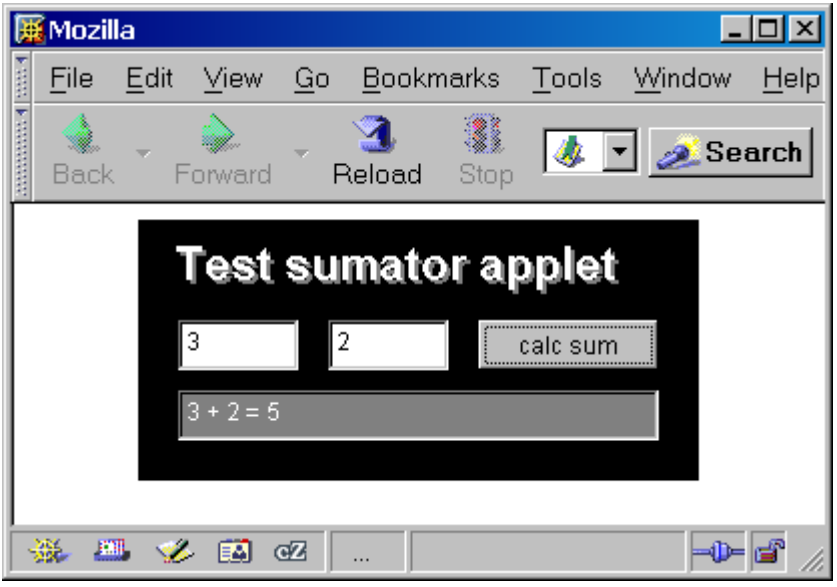
Ето примерен HTML код, с който може да се стартира аплета:

TestSumatorApplet.html

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

Забележете параметъра **codebase="."**, с който се задава пътя до директорията, в която се намира **.class** файла на аплета. Без него аплетът не може да се изпълни, защото реално се състои от повече от един **.class** файлове.

Ето как изглежда нашият аплет в браузъра Mozilla:



Тагът **<applet>** има и други интересни параметри, като например **archive="SomeArchive.jar"**, с който може да се зададе пътя и името на JAR архив, който съдържа класовете на аплета.

Използване на JAR архиви

За да демонстрираме използването на JAR архиви ще компилираме и пакетираме в JAR архив `.class` файловете на аплета `SumatorApplet`. Ето един пример как може да стане това:

compile_sumator_applet.cmd

```
del *.class
javac SumatorApplet.java
del SumatorApplet.jar
jar -cvf SumatorApplet.jar *.class
```

Изпълняването на аплета от JAR файла може да стане със следния HTML код:

TestSumatorApplet-JAR.html

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

Библиотеката Swing

Преди да изясним ситуацията с правата на аплетите, трябва да отбележим, че в новите версии на JDK след 1.1 съществува стандартно разширение на библиотеката **AWT**, което се нарича **Swing** и представлява съвкупност от класове за създаване на прозоречно-ориентиран графичен потребителски интерфейс, които се намират в пакета `javax.swing`.

Ние няма да разглеждаме библиотеката Swing, но тези от вас, които искат да я използват, трябва да знаят, че тя изисква инсталиран Java Plug-In версия 1.2 или по-висока.

Таговете <embed> и <object>

Досега в нашите примери за да изпълним аплет използвахме тага `<applet>`. Въпреки, че това е най-лесният начин да вградим аплет в дадена HTML страница, той не е много препоръчителен. Проблемът на `<applet>` тага е, че при него не може да се укаже коя версия на JDK изисква аплетът за да работи нормално. Друг проблем е, че ако браузърът не поддържа аплети, потребителят въобще няма да бъде уведомен за това и няма да бъде автоматично помолен да си инсталира Java Plug-In.

За решаването на тези проблеми могат да бъдат използвани таговете **<object>** в Internet Explorer и **<embed>** във всички останали браузъри. В тях може да се укаже минималната версия на JDK, която е необходима на аплета, както и от къде може да бъде изтеглен Java Plug-In ако такъв няма на машината на клиента. Понеже Internet Explorer не разбира тага **<embed>**, а останалите браузъри не разбират тага **<object>**, е необходимо тези два тага да се комбинират, за да се направи аплетът да работи в средата на всички популярни Web-браузъри като се съобразява с поисканата версия. Ето един пример как може да стане това:

BallAppletNewTest.html

```
<html>
<head><title>Ball Applet - New Test Page</title></head>
<body>
<object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
codebase="http://java.sun.com/products/plugin/autodl/jinstall-
1_4-windows-i586.cab#Version=1,4,0,0" width="200" height="150">
    <param name="code" value="BallApplet.class">
    <param name="type" value="application/x-java-
applet;version=1.4">
    <param name="imageName" value="ball.jpg">
    <comment>
    <embed type="application/x-java-applet;version=1.4"
        code="BallApplet.class" width="200" height="150"
        imageName="ball.jpg"
pluginspage="http://java.sun.com/products/plugin/index.html#down
load">
        <noembed>
            Applet can not be started because
            Java Plug-In 1.4 is not installed.
        </noembed>
    </embed>
    </comment>
</object>

</body>
</html>
```

При зареждане на тази страница в Internet Explorer, ако на машината няма инсталиран Java Plug-In, той се издърпва автоматично, като ActiveX контрола и след потвърждение от потребителя се инсталира. При други браузъри инсталирането на Java Plug-In (ако го няма или е много стара версия) не става автоматично, но браузърът се пренасочва автоматично към страницата за изтеглянето му.

Не е необходимо за знаете точния синтаксис на тези тагове, защото за автоматичното преобразуване на тага **<applet>** към по-новите тагове

стандартно към JDK 1.4 има включена специална помощна програмка **`HtmlConverter.exe`**, която е достъпна от **`bin`** директорията на JDK. С нея лесно можете да преобразувате **`<applet>`** тага към по-сложни тагове за извикване на аплет, които са съобразени с различни Web-браузъри.

2.3. Java аплети и сигурност. Комуникация със сървъра

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.

За решаването на този проблем аплетите са проектирани да 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.

Съществува и друга стандартна възможност – аплетите да се подписват цифрово. Подписаните аплети могат да се изпълняват без ограничения на правата, но само ако при зареждането им потребителят им разреши това. Подписаните аплети доказват пред потребителя че са безопасни чрез цифров сертификат, използван при подписването им. Ако потребителят вярва на сертификата, той може да се съгласи да ги изпълни без ограничения на правата, а в противен случай, ако не им вярва, те се изпълняват както обикновени аплет.

Какви права имат аплетите

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 осъществяват 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. На практика последното понякога 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.

Аплетите, сигурността и комуникацията със сокети

Трябва да обърнем специално внимание на сокетите. Свидетели сме на много аплети, които извършват активна мрежова дейност, като например аплети за разговори (chat), аплети за четене на e-mail, аплети за изпращане на e-mail, различни игри и др. 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 получат 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.

Много удобен за такива ситуации е TCP forwarding сървърът, който представихме в предходната глава. Чрез него можем да преодолеем ограничението, наложено на аплетите, да отварят сокети само към хоста, от който са заредени и можем да му позволим да комуникира със всеки друг хост. При използването на този подход много трябва да се внимава със сигурността, защото всеки TCP forwarding сървър, работещ на машина, видима от Интернет лесно може да стане обект на атака от злонамерени лица, които искат да го използват за нелегална дейност, например за разпращане на спам.

Пример за аплет, който комуникира със сървъра

Ще дадем един пример, който илюстрира как можем да реализираме аплет, който асинхронно получава данни от сървъра. Да си представим,

че искаме да направим сайт, от който да се следят цените на акциите на големи компании на борсата. Цената на акциите на борсата е нещо, което много бързо се променя (всяка секунда или дори по-често). Искаме постоянно да визуализираме актуалната в момента цена за дадена компания.

Да предположим, че имаме TCP сървър, който приема идентификатор на компания (company ticker), след което започва постоянно да изпраща информация за цената на акциите на тази компания при всяка нейна промяна. Искаме да направим аплет, който отваря един TCP сокет, изпраща по него към сървъра идентификатор на компания и след това постоянно отпечатва всичко, което прочете от сокета. Една примерна реализация може да изглежда по следния начин:

StockQuoteApplet.java

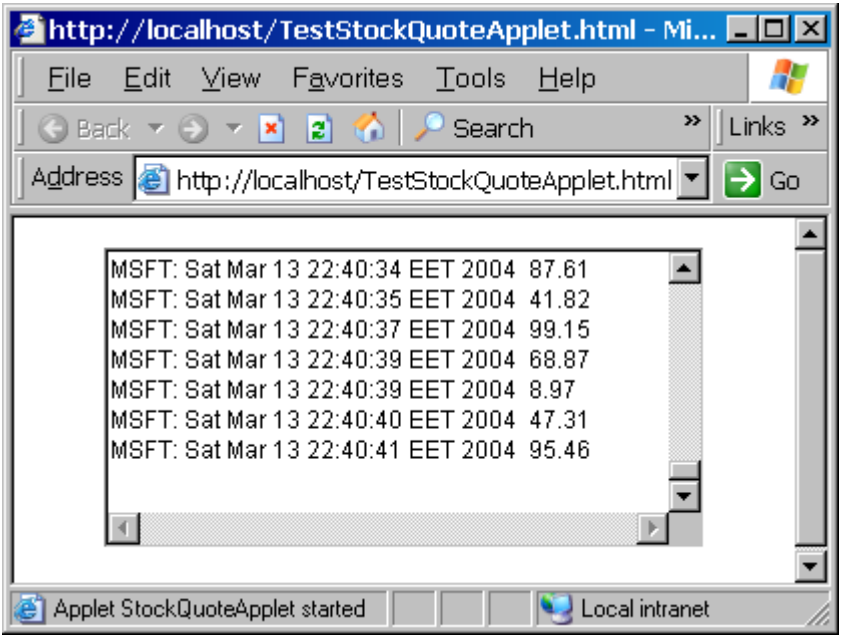
```
import java.applet.Applet;
import java.awt.*;
import java.net.Socket;
import java.io.*;

public class StockQuoteApplet extends Applet
        implements Runnable {
    public static final int STOCK_SERVER_PORT = 2004;
    public static final String COMPANY_TICKER = "MSFT";

    private BufferedReader mSocketReader;
    private TextArea mTextArea = new TextArea();

    public void init() {
        try {
            // Establish TCP socket connection with the server
            String host = this.getCodeBase().getHost();
            Socket sock = new Socket(host, STOCK_SERVER_PORT);

            // Send the company ticker to the server
            OutputStreamWriter socketWriter =
                new OutputStreamWriter(sock.getOutputStream());
            socketWriter.write(COMPANY_TICKER + "\n");
            socketWriter.flush();

            // Get the input stream reader
            mSocketReader = new BufferedReader(
                new InputStreamReader(sock.getInputStream()));
        } catch (IOException ioex) {
            ioex.printStackTrace();
            System.exit(-1);
        }

        // Set the layout manager to null
```

```
        this.setLayout(null);

        // Create the text area and add it to the applet
        mTextArea.setBounds(new Rectangle(0, 0, 300, 150));
        this.add(mTextArea);

        // Create and start socket reader thread
        Thread sockerReaderThread = new Thread(this);
        sockerReaderThread.start();
    }

    public void run() {
        try {
            while (true) {
                String line = mSocketReader.readLine();
                mTextArea.append(line);
                mTextArea.append("\n");
            }
        } catch (IOException ioex) {
            ioex.printStackTrace();
        }
    }
}
```

Аплетът отваря сокет към сървъра по време на инициализацията си, след което изпраща по сокета идентификатор на компания (в случая MSFT) и стартира отделна нишка, която да чете постоянно данните идващи от сокета и да ги добавя в текстовата област. Нужда от синхронизация не е необходима, защото текстовата област се променя само от една нишка и не може да стане конфликт.

Изглед от аплета за визуализация на акциите на борсата

Ето как изглежда аплета в действие (в Internet Explorer 6):



Ето и примерен HTML код, с който може да се изпълни този аплет:

TestStockQuoteApplet.html

```
<html><body><center>
<applet code="StockQuoteApplet.class" codebase="."
    width="300" height="150">
</applet>
</center></body></html>
```

Остава да предложим и примерна реализация на сървъра за следене на акциите на борсата. За леснота ще изпращаме на клиентите случайни данни с пълното съзнание, че са фалшиви. В реална ситуация данните биха могли да се извличат от някаква база данни или от друг сървър. Ето как би могъл да изглежда сорс-кодът:

StockQuoteServer.java

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

public class StockQuoteServer {
    public static int PORT = 2004;
```

```
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(PORT);
        while (true) {
            Socket socket = serverSocket.accept();
            StockQuoteThread clientThread =
                new StockQuoteThread(socket);
            clientThread.start();
        }
    }
}

class StockQuoteThread extends Thread {
    private Socket mSocket;
    private BufferedReader mSocketReader;
    private PrintWriter mSocketWriter;
    private Random mRandomGenerator = new Random();

    public StockQuoteThread(Socket aSocket) throws IOException {
        mSocket = aSocket;
        mSocketReader = new BufferedReader(
            new InputStreamReader(mSocket.getInputStream()));
        mSocketWriter = new PrintWriter(
            new OutputStreamWriter(mSocket.getOutputStream()));
    }

    public void run() {
        try {
            String companyTicker = mSocketReader.readLine();
            while (!isInterrupted()) {
                String quote = companyTicker + ": " +
                    getDate() + " " + getRandomQuote();
                mSocketWriter.println(quote);
                mSocketWriter.flush();
                int delay = mRandomGenerator.nextInt(3000);
                Thread.sleep(delay);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private String getRandomQuote() {
        int value = mRandomGenerator.nextInt(10000);
        return " " + (value / 100) + "." + value % 100;
    }

    private String getDate() {
        return (new Date()).toString();
    }
}
```

Бъдещето на аплетите

В Интернет е пълно със сайтове, които активно използват аплети. Типичен пример за такъв сайт е ICQ2Go – Web клиентът за ICQ достъпен от адрес http://go.icq.com/. Макар и да изискват предварително инсталиране на Java Plug-In за да стартират много такива сайтове отбелязват забележителен успех. Аплетите решават най-разнообразни проблеми, за които няма друга стандартна алтернатива. Едно от най-ценните им качества е, че работят на всички популярни платформи, за които има Web-браузър с поддръжка на Java.

Липсата на стандартна поддръжка за по-високи версии на JDK от 1.1 в повечето браузъри ни кара да се замислим дали аплетите не са остаряла и леко поизоставена технология. Възниква въпросът защо ако са толкова хубаво нещо не се поддържат стандартно във всеки браузър, така както във всеки браузър се поддържа Flash. Една от причините е решението на Microsoft да не поддържа стандартно Java в Internet Explorer, а този Web-браузър е най-масово използваният в света и влиянието му е наистина огромно.

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-базирани приложения.

Алтернативна технология, която е масово навлязла, е Macromedia Flash, която е предназначена главно за създаване на мултимедийни сайтове. Тя, обаче, няма същата мощ, която има Java и поради това не може да замести аплетите.

Амбицията на Microsoft да замени Java аплетите с client-side .NET Windows Forms контроли може да компенсира в известна степен отдръпването им, но не напълно заради платформената си зависимост. Засега Windows Forms контролите работят само под Windows с Microsoft Internet Explorer.

Друга платформено-зависима алтернатива на аплетите са ActiveX контролите. При тях проблемите са дори повече – освен че работят само под Windows, те изискват потребителят да им се довери, т.е. нямат механизъм който да ограничи правата, с които се изпълняват, което ги прави много опасни за сигурността.

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» ***Светлин Наков*** е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» ***Мартин Кулов*** е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Глава 3. Разработка на Web-приложения с Java

Web-приложенията представляват софтуерни системи, които са достъпни през Интернет или локална мрежа чрез стандартен Web-браузър. Например всички Web-базирани системи за електронна поща (като например **mail.yahoo.com**, **abv.bg** и **mail.bg**) представляват Web-приложения.

За достъп до една Web-базирана система, е необходимо потребителят да разполага със стандартен Web-браузър (например Internet Explorer или Mozilla) и връзка до машината, 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.

Както знаем, 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.

В настоящата глава ще навлезем в Java технологиите за изграждане на Web-приложения. Ще се запознаем с основните концепции на Web-програмирането и протокола HTTP, след което ще представим технологията „Java сървлети". Ще продължим с преглед на сървъра за Web-приложения Tomcat. Ще се запознаем с начините за предаване на информация от клиента към сървъра чрез HTML форми, след което ще разгледаме жизнения цикъл на сървлетите и ще продължим с представяне на средствата, които Java Servlet API предоставя за автоматизирана поддръжка на потребителски сесии. Накрая ще се запознаем с технологията Java Server Pages (JSP) и ще представим цялостен пример за Web-приложение.

3.1. Основни понятия. Web-сървър. Протокол HTTP

Web-сървър

За публикуването на информация в Web пространството (WWW) се използват Web-сървъри. Web-сървърът представлява софтуер, който предоставя достъп до някаква информация по протокол HTTP. Web-сървърите могат да предоставят както статични ресурси, така и ресурси, които се създават в момента на изпълнение на заявката за достъп до тях (динамично генерирана информация).

Web-сайт

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 и на няколко Web-сървъра.

Web-приложения

Едно Web-приложение представлява софтуерна система с Web-базиран потребителски интерфейс, работеща на някакъв Web-сървър в Интернет или 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 резултати от няколко независими Web-приложения, работещи на различни и отдалечени един от друг сървъри да се визуализират на 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 от тях 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 посетители, 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.), динамично 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 Web-приложенията могат да бъдат разпределени на няколко сървъра и да работят като единна система.

Най-често под Web-приложение се разбира цялостна софтуерна система, която служи 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.

Особености на Web-приложенията

Характерна черта за Web-приложенията е, че към тях едновременно осъществяват достъп много потребители. Всеки потребител се обслужва независимо от другите потребители, така сякаш е единствен.

Друга характерна черта на Web-приложенията е, че работят с еднопосочна комуникация, по модела 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 осъществяват визуализиране на информация в реално време.

Web-приложенията и езиците за програмиране

Web-приложения могат да се разработват на различни езици и за различни Web-платформи – CGI, Perl, PHP, ASP, ASP.NET, Java/JSP и още много други. 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, стандарти и библиотечни функции) за разработка на Web-приложения, който ще разгледаме по-късно. Нека започнем с някои основни понятия от света на WWW.

Какво е Web-сървър

От гледна точка на Интернет програмирането Web-сървърите са приложения, които “слушат” на определен TCP порт (обикновено това е

стандартният порт за протокола 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 клиентски заявки файловете, които са разположени в дадена директория, обозначена като главна Web-директория. Например ако имаме един прост Web-сървър стартиран на машината с име **`www.mywebserver.com`** и сме указали, че главната му директория е **`C:\MyWebSite`**, то когато даден Web-браузър поиска ресурс по даден URL от този сървър, например **`http://www.mywebserver.com/pictures/index.html`**, нашият прост Web-сървър ще му предостави файла **`C:\MyWebSite\pictures\index.html`** (ако съществува).

Common Gateway Interface

Всички съвременни Web-сървъри имат възможността да предоставят на клиентите си не само файлове от главната си Web-директория и нейните поддиректории, но и динамично генериран HTML, получен от работата на външна за Web-сървъра програма. Технологията, при която динамичното съдържание се генерира от външна за сървъра програма, се нарича CGI (Common Gateway Interface).

При CGI на базата на HTTP заявката Web-сървърът стартира някоя външна за сървъра програма (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 за програмиране или script, например на C, C++, Pascal, Perl, PHP и др.

Други технологии за генериране на динамично съдържание

Има и други възможности за динамично генериране на HTML, които не се базират на CGI – например не чрез външна програма, а чрез модул вграден директно в 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.

Друга широко използвана възможност да се генерира динамично Web съдържание е като се използват скриптови езици като VBScript, Perl и PHP. При тях не се използва компилиран код, а динамичното

съдържание се генерира, като по време на изпълнение на заявката скриптът, който я обработва, се изпълнява от някакъв интерпретатор, който е вграден по някакъв начин в сървъра. Всъщност в по-новите си версии езиците Perl и PHP не работят с интерпретатор, а с компилатор и виртуална машина, която изпълнява компилирания код. Чрез добавяне на специални модули виртуалните машини на Perl и PHP могат да бъдат вградени в Web-сървъра и да изпълняват PHP и Perl код директно в процеса на сървъра. Такива са например модулите **mod_perl** и **mod_php** за Web-сървъра **Apache**.

Вграждането на компилиран програмен код има някои предимства пред извикването на външна програма, заради което CGI технологията се счита за остаряла и се използва все по-рядко. По скорост на изпълнение е по-ефективно да се използва вграден в сървъра код, защото не се налага при всяка динамична заявка да се извиква външно приложение, което е свързано със създаването на нов процес в операционната система. Друго предимство е, че интеграцията между сървъра и вградения в него компилиран код е по-силна и по-лесна за реализиране отколкото интеграцията с външна CGI-програма. При CGI е по-трудно да се реализира автоматизирана поддръжка на потребителски сесии, както и frameworks за Web-приложения.

Протоколът HTTP

Не е редно да се впускаме в света на Web-програмирането преди да сме изяснили в детайли протокола, по който си комуникират Web-сървърите с Web-браузърите – HTTP (Hyper Text Transfer Protocol).

HTTP представлява прост текстов протокол, който се използва от услугата WWW за осигуряване на достъп до практически всякакъв 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 върху стандартен TCP сокет отворен от клиента към сървъра. Стандартният порт за HTTP протокола е 80, но може да се използва и всеки друг 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.

В развитието си HTTP протоколът е преминал през версиите 0.9, 1.0 и 1.1, която е най-разпространена. На практика когато се говори за HTTP, обикновено се има предвид HTTP 1.1.

HTTP заявки

HTTP заявката при версия 1.1 на протокола има следния формат:

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

HTTP заявките имат 3 основни елемента: **метод**, **Request-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 и извличане на ресурс. Имената на методите в HTTP заявките се изписват винаги с главни букви.

Идентификаторът **`Request-URI`** определя ресурса, над който ще оперира заявката. В частта Request-URI могат да се използват два вида идентификатори:

- **URI идентификатор** (Uniform Resource Identifier)
- **релативен път** спрямо главната директория на Web-сървъра

Един URI идентификатор може да бъде или URL адрес (Uniform Resource Location, например http://www.nakov.com/inetjava/index.html), т.е. да е идентификатор на ресурс, зададен чрез уникалното си местоположение или URN име (Uniform Resource Name, например **`urn:isbn:954-8905-06-X`**), т.е. да е идентификатор на ресурс, зададен чрез уникалното си име по даден URN namespace идентификатор (за нашия пример това е идентификатора isbn). В практиката URN схемата за идентификация на ресурс почти не се използва при HTTP заявки.

Релативният път спрямо главната директория на Web-сървъра задава местоположението на ресурс в рамките на текущия Web-сървър. Това е частта от URL, която стои след името на хост-а (сървъра) в URL идентификатора. Например един такъв релативен път може да бъде идентификаторът **`/inetjava/index.html`**.

Фрагментът **HTTP/1.1** с който завършва първият ред на HTTP заявката задава версията на HTTP протокола, която ще бъде използвана.

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.

Примерна HTTP заявка

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
↵
```

Как се отличават виртуалните хостове на един Web-сървър

На един Web-сървър може да има хостнати няколко различни сайта, които могат да започват с различни Интернет имена. Такива сайтове се наричат виртуални хостове в рамките на Web-сървъра. Например сайтовете с URL адреси http://www.nakov.com/ и http://bgcon.org/ могат да са хостнати в Web-сървъра на една и съща машина, да кажем машината с IP адрес 194.12.244.90. Възниква въпросът как браузърът указва кой от двата адреса иска, след като те се обслужват от един и същ Web-сървър на една и съща машина.

Ясно е, че в услугата DNS Интернет имената www.nakov.com и bgcon.org трябва да са регистрирани да съответстват на IP адреса 194.12.244.90. Когато в полето за адрес на един стандартен Web-браузър се напише един от горните два URL адреса, да кажем http://www.nakov.com/, браузърът прави следното: Първо чрез услугата DNS получава IP адреса на машината, която хоства търсения сайт (www.nakov.com). След това отваря TCP връзка към тази машина на порт 80 и изпраща заявка за извличане на ресурса „/". В хедъра на тази заявка браузърът указва в полето Host стойността www.nakov.com. Именно по това поле Host в хедъра на HTTP заявката Web-сървърът разбира за кой от всички виртуални хостове се отнася тази заявка.

Другият начин при HTTP заявка да се укаже виртуалният хост в Web-сървъра е да се използва URI идентификатор на искания ресурс (URL адрес). В този случай името на хоста се включва в самия този идентификатор.

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 значение за Web-програмирането.

HTTP GET заявки

GET методът представлява команда за извличане на ресурс, указан от зададено URI или релативен път в рамките на Web-сървъра. 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 изпълнява програмния 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`** от директорията на виртуалния хост **`inetjava.sourceforge.net`**.

Тази заявка съвсем истинска. Тя е генерирана от Web-браузъра Internet Explorer 6.0 при опит да се отиде на URL адрес http://inetjava.sourceforge.net/InetJava-2002-program.html и е прихваната чрез софтуер за подслушване на мрежовия трафик.

При HTTP GET заявката 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 стане дума по-нататък.

HTTP POST заявки

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 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 адрес **`http://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 (заедно със символите за край на ред). 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 на параметрите и стойностите им се прави автоматично от Web-браузъра, а разкодирането им съответно от скрипта, който обработва заявката на Web-сървъра. Като Web-програмисти на Java не е необходимо да знаем в детайли точно как става това разкодиране, защото то се извършва автоматично от средата, в която се изпълнява нашето Web-приложение (т. нар. Web-контейнер).

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 код на грешка заедно с текстово описание защо се е получила. 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 **ресурсът**, кодиран по описания в header-полетата 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 за 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 при кеширане на документи); 404 – ресурсът не е намерен; 500 – грешка на сървъра. Ето един цялостен 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 е 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>
```

Вижда се, че сървърът освен че връща в отговор на заявката код на грешка 404 изпраща и допълнителна информация, която пояснява значението на този код. Тази информация е във вид на HTML документ, защото е предназначена да бъде показана в браузъра на клиента, който е поискал липсващия ресурс.

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 протокола по-бърз заради което е предпочитан от повечето HTTP клиенти.

3.2. Основни концепции в Web-програмирането

В настоящата тема ще се опитаме да представим различните аспекти на Web-програмирането, неговите предимства и недостатъци, както и причините за неговата популярност.

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, Web-приложенията представляват програмни системи, които работят на някакъв Web-сървър и предоставят на потребителите Web-базиран интерфейс, който се визуализира от Web-браузърите им. Комуникацията между потребителските Web-браузъри 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.

Архитектура на Java-базираните Web-приложения

Можем да разделим Java-базираните Web-приложения на две-части:

- **сървърска част** – представлява съвкупност от Java сървлети и JSP страници, които обработват получените от потребителя данни и в зависимост 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 на потребителя 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.

Web-контейнер

Всяко Web-приложение се изпълнява от някакъв сървър. Например приложенията написани на PHP най-често се изпълняват на Web-сървър **Apache**, към който е включен специален модул за поддръжка на PHP – **mod_php**. Приложенията написани с технологията ASP.NET обикновено се изпълняват на сървър **IIS** (Microsoft Internet Information Server), към който е включен специален модул за изпълнение на ASP.NET

приложения. По същия начин приложенията написани със средствата на Java се изпълняват от някакъв сървър за Java Web-приложения, който се нарича още **Web-контейнер** или **Servlet-контейнер**.

При Java-базираните Web-приложения Web-контейнерът обикновено служи като Web-сървър, който може да доставя на клиентите както статично съдържание, така и динамично съдържание, получено при изпълнението на Java сървлети и JSP страници. Понякога като Web-сървър се използва отделно приложение, което препраща на Web-контейнера заявките за изпълнение на сървлети и JSP страници.

Servlet-контейнерът осигурява среда за изпълнение на сървлети и JSP страници. Той осигурява инфраструктурата, специфицирана в J2EE, която е необходима за нормалната работа на сървлетите и JSP страниците. Той се грижи заявките, идващи по протокол HTTP да се подават на съответния сървлет или JSP, той се грижи за автоматичното получаване на параметрите, подадени от клиента, той се грижи за автоматичното управление на сесии и за още много други неща, описани в Servlet и JSP спецификациите.

Примери за Java Web-контейнери (Servlet-контейнери) са популярните сървъри с отворен код Tomcat и Jetty.

Според спецификацията за J2EE всеки J2EE сървър трябва да съдържа в себе си JSP/Servlet контейнер (Web-контейнер), но J2EE сървърите за приложения (J2EE application servers) включват освен Web-контейнер още много други неща, описани в J2EE спецификацията (например EJB контейнер, JMS имплементация, поддръжка на Web услуги и т.н.). Примери за J2EE сървъри са JBoss, BEA WebLogic, IBM WebSphere и Oracle OC4J. Всички те поддържат сървлети и JSP страници и могат да изпълняват Java-базирани Web-приложения.

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 описание на документи, широко използван от в миналото от американското правителство) и наследява до известна степен 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 и 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), които могат да сочат към произволни отдалечени ресурси, зададени чрез URL. По тези връзки потребителите могат лесно да преминават от един документ към други документи без предварително да знаят точните им адреси.

CSS

Cascading Style Sheets е допълнение към HTML. Разработен е от W3C (World Wide Web Consortium) и представлява език за описание на начина на визуализация и позициониране 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 елементите на визуализираните в браузъра HTML документи. Програмният код, написан на JavaScript, се вгражда в HTML документите или се записва във външен файл и се изпълнява от интерпретатора на JavaScript при определени събития, възникнали при взаимодействието на потребителя с визуализирания в браузъра документ или при събития породени от самия браузър. С помощта на JavaScript е възможно създаването на сложни по функционалност интерактивни Web-сайтове.

От съображения за сигурност JavaScript има достъп само до ресурси, които са свързани с визуализираните в момента HTML документи в Web-браузъра. От JavaScript не можем да отваряме файлове, не може да използваме сокет връзки, не можем да комуникираме с други приложения освен с Web-браузъра, който изпълнява JavaScript кода и не можем да извършваме много други действия, които застрашават личните данни на потребителя.

DHTML

Съвкупността от технологиите HTML, CSS и JavaScript се нарича DHTML (Dynamic HTML). DHTML предлага богати възможности за реализиране на потребителския интерфейс на сложни Web-приложения. Въпреки това винаги, когато разработваме Web-приложения, трябва да се съобразяваме с факта, че потребителският интерфейс на Web-

приложенията е ограничен и не може да се равнява на прозоречно-ориентирания графичен потребителски интерфейс, който е достъпен в Win32, Java или .NET базираните приложения. Има някои елементи на потребителския интерфейс, които не могат да се реализират с DTHML, а други могат да бъдат реализирани, но се правят много трудно с

Съвместимост между различните Web-браузъри

От създаването си до днешни дни технологиите HTML, JavaScript и CSS претърпяха сериозно развитие. В началните години на навлизането на Web-технологиите като стандартен начин за предоставяне на информация в Интернет имаше сериозни разминавания в разбиранията на различните производители на Web-браузъри относно това как да се визуализира HTML. Съвсем нормално беше една Web-страничка се визуализира правилно и да работи добре с Internet Explorer, а да не работи с Netscape Navigator или обратното. Беше се стигнало до там, че някои сайтове поддържаха по няколко съвсем отделни версии за различните браузъри. Стандарти имаше, но не се спазваха, а всеки производител добавяше допълнителни възможности към своя браузър, заради което имаше много сериозни проблеми при съвместимостта както при HTML, така и при JavaScript и CSS.

Няколко години след това стандартите започнаха да стават по-строги, а производителите на Web-браузъри започнаха да ги спазват по-стриктно и в крайна сметка съвместимостта между различните браузъри сериозно се подобри. От тогава сайтовете, които спазват стриктно стандартите за HTML, JavaScript и CSS започнаха да изглеждат почти еднакво на всички водещи Web-браузъри.

Разпределеност и платформена независимост на Web-приложенията

Най-голямото предимство на Web-програмирането е, че Web-приложенията са достъпни от всеки, които има връзка с Интернет и стандартен Web-браузър. Това означава, че написано веднъж едно Web-приложение може да се използва от различни компютри, работещи под различни операционни системи с различни браузъри, като единственото условие е те да имат достъп до Web-сървъра, където работи съответното Web-приложение. На практика Web-технологиите отварят огромни възможности за лесно създаване на многопотребителски разпределени приложения.

Поради независимостта от операционната система на потребителя и относителната независимост от неговия Web-браузър Web-програмирането значително 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.

Java разработчиците могат да се възползват и от платформената независимост на езика Java. Благодарение на нея не само клиентската, но и сървърската част на Web-приложенията става платформено независима. Java-базираните Web-приложения освен независими от операционната система са независими и от производителя на сървърски софтуер, в средата на който се изпълняват (Web-контейнер или J2EE application server). Допълнително, благодарение на унифицирания Java стандарт за достъп до бази данни JDBC е възможно в голяма степен да се осигури и независимост от производителя и при използване на бази данни.

Несесийност на HTTP протокола

По идея Web-приложенията 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 още много други неща, които не са били предвидени при създаването му и за които не е толкова удобен. Въпреки, че не е много подходящ за някои, HTTP се използва за тях, защото се е наложил като стандарт. Въпреки че версия 1.1 на HTTP въвежда така наречените keep-alive връзки (възможност за изпълнение на няколко HTTP заявки една след друга през веднъж отворен съществена връзка между клиента и сървъра), това не решава проблема с еднозначната идентификация на потребителя от страна сървърската част на Web-приложението. Обикновено за целта се използват допълнителни техники, като т. нар. **cookies** и други похвати, с които се идентифицират и разпознават отделните потребители.

За щастие средствата за осигуряване на многопотребителски достъп са вградени стандартно в J2EE, 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 Web-приложения.

Ниски изисквания към клиента

Ниските изисквания към клиента са един от големите плюсове на Web-програмирането. При този вид програмиране бизнес операциите на системата са изнесени на сървъра, 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.

Сигурност от страна на клиента

В компютърната индустрия вируси и други злонамерени програми, които се разпространяват с изпълнимите програми, са нанесли големи 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 крайния потребител, тъй като при тях клиентската част се състои във визуализирането на документи, описани с HTML/CSS, и изпълнението на JavaScript, а тези технологии не могат да навредят на данни и другите ресурси, с които потребителя разполага.

Сигурност 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 и др.), без да могат да го достъпват директно, а само посредством някое Web-приложение. Съответното Web-приложение може да дава достъп до определени ресурси само след автентикация на потребителя с парола, клиентски сертификат или друга технология. Въвеждането на HTTPS като стандарт прави на практика невъзможно подслушването на конфиденциална информация, обменяна между Web-приложение и клиентски браузър.

Недостатъци на Web-програмирането

Проблем за много Web-приложения е производителността. Бавната връзка между клиента и сървъра води до ниска производителност на цялата система. Дори и сървърът да обработва и отговаря мигновенно на клиентските заявки, потребителите често имат усещането, че системата, с която работят, е бавна, заради забавянето, 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.

Друг 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 програмистът вложи 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 в Web-среда. 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 основано на модела "заявка-отговор", който не позволява 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 разпределени, в които комуникацията между клиента и сървъра е възможна и в двете посоки.

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.

Друга причина е платформената независимост на клиента, която вече разгледахме.

3.3. Java базирани Web-приложения

Едно Java базирано Web-приложение представлява съвкупност от сървлети, JSP страници (разширение на сървлетите, което позволява в HTML документи да се вгражда Java код), Java архиви и други файлове, които заедно изграждат една обща Web-базирана система.

Производителността на сървлети и JSP

От гледна точка на ефективността сървлетите и JSP-тата превъзхождат вече остарялата CGI технология, защото изпълнението на сървлет (или JSP) 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 от нея и резултатът се връща на клиента, без да се създава нов процес за обработка на заявката.

Веднъж изпълнен, сървлетът остава в компилиран вид активен в паметта, чакайки ново извикване. Това значително повишава производителността, защото не е необходимо при всяка заявка сървлетът да се чете от файловата система, да се инстанцира неговият клас. Когато едновременно постъпят няколко заявки за един и същ сървлет, той се изпълнява едновременно в няколко нишки, за което се грижи сървъра.

Всъщност по подобен на сървлетите начин работят всички съвременни Web-технологии като Perl, PHP, ASP и ASP.NET. 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. За по-голяма ефективност често се прилага и друг подход – не се използва интерпретатор, а компилатор, който компилира кода до някакъв междинен код и виртуална машина, която изпълнява този междинен код.

Средствата на J2EE за изграждане на Web-приложения

Съвкупността от средства 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 сървлетите и JSP-тата).

Преносимост на Java-базираните Web-приложения

Java базираните Web-приложения са 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.

Сигурност и надеждност на Java-базираните Web-приложения

Сигурността и надеждността на Java базираните Web-приложения е завидно добра. Това се дължи най-вече на надеждността и сигурността на самата Java платформа. 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 много по-рядко срещани, отколкото при други платформи.

На практика тенденцията е повечето езици за програмиране с общо предназначение да преминават към архитектури, близки до архитектурата на Java. Компютрите станаха толкова бързи, че вече използването на виртуална машина, интерпретатор или някаква друга среда, която управлява изпълнението на кода и не му позволява да прави „лоши неща” се е наложило като стандартен модел за изпълнение на програмния код. Такъв е моделът на изпълнение и в Java и в .NET и в Perl и в PHP и в други платформи.

Езиците за програмиране в платформите Java и .NET имат и други предимства по отношение на сигурността и надеждността – те са силно типизирани, а това не позволява възникване на проблеми с неправилно използване на указатели. В тези езици има и автоматично управление на паметта и ресурсите, а това силно ограничава проблемите със сигурността и надеждността, свързани с неправилно заделяне, освобождаване на памет и управление на ресурси.

Java-базираните Web-приложения и свободния софтуер

Още едно важно предимство на Java сървлетите, JSP страниците и Java-базираните Web-приложения е, че те могат да се използват напълно безплатно, включително и в комерсиални проекти. 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), идеални за малкия и средния бизнес. Примери за такива безплатни сървъри са Web-контейнерът Tomcat (http://jakarta.apache.org/tomcat/), който ще разгледаме след малко и сървърът JBoss (http://www.jboss.org), който е пълна имплементация на J2EE спецификацията.

3.4. Java сървлети

Java платформата предоставя няколко стандартни средства за създаване на динамични Web-страници. Основната технология, на базата на която се изгражда всичко останало, са **Java сървлетите**.

Какво са Java сървлетите

Java 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.

Пример за сървлет

Да дадем пример за прост HTTP сървлет, който при извикване връща текущата дата и час, форматирани като прост HTML документ. Противно на всички книги за компютри нашият първи пример не е сървлетът „Hello, world!". Ето как изглежда нашият код:

FirstServlet.java

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

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

Всичко, което прави примерният сървлет **`FirstServlet`**, 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.

Как се създава Java сървлет

За създаването на Java сървлет за обслужване на Web-заявки е необходимо да се наследи класа **javax.servlet.http.HttpServlet** и да се припокрие един от методите **doGet()** или **doPost()** съответно в зависимост дали сървлетът ще обработва съответно GET или POST HTTP заявки.

И двата метода **doGet()** и **doPost()** приемат като параметри 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 и позволява задаване на HTTP кода на резултата и полетата в хедъра на HTTP отговора, както и 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, извличане на параметри, управление на потребителската сесия, работа с cookies и други специфични действия, 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**. Ние ще разглеждане само HTTP сървлети и под думата сървлет ща разбираме винаги HTTP сървлет.

Как се изпълнява Java сървлет

За да изпълним нашият примерен сървлет са необходими няколко стъпки:

1) компилираме сървлета и получаваме съответен .class файл;
2) инсталираме и конфигурираме някакъв Servlet-контейнер, в средата на който ще се изпълнява сървлета;
3) deploy-ваме сървлета в сървлет-контейнера (инсталираме го и го подготвяме за изпълнение):
   - създаваме Web-приложение в сървлет-контейнера;
   - копираме сървлета в директория **WEB-INF/classes** на това Web-приложение;

- регистрираме сървлета в конфигурационния файл на Web-приложението **web.xml**;

4) стартираме Servlet-контейнера и извикваме сървлета чрез заявка от стандартен Web-браузър.

Как се използва Servlet -контейнера Tomcat ще разгледаме след малко, но преди това нека обърнем внимание на една особеност, с която ще се сблъскаме при компилирането – липсата на някои класове, които използваме.

Къде са дефинирани класовете javax.servlet.*

Класовете от пакетите **javax.servlet** и **javax.servlet.http** не са стандартна част от JDK. За да ги използваме, е нужно да включим към проекта си библиотеката Servlet API, в която са дефинирани тези класове. Обикновено тази библиотека се разпространява заедно със Servlet-контейнера. Например при Tomcat 3.x в директория **lib/common** има файл с име **servlet.jar**, който съдържа тези класове. При Tomcat 4.x файлът **servlet.jar** се намира в директория **common/lib**, а при Tomcat 5.x този файл се казва **servlet-api.jar** и се намира в директорията **common/lib** на Tomcat. При други Web-контейнери и Web application сървъри Servlet API библиотеката може да се намира на други места.

Ако искаме да компилираме сървлет и компилаторът не намира класовете от пакета **javax.servlet**, трябва да си включим към проекта или към клас-пътя на компилатора Servlet API библиотеката. Последната версия на тази библиотека може да се изтегли от адрес http://java.sun.com/products/servlet/downloads/.

3.5. Работа със сървъра Tomcat

Сървърът Tomcat представлява безплатен Web-контейнер, който може да изпълнява Java сървлети, JSP страници и Web-приложения. Tomcat е Web-application сървър, написан на Java, който при клиентска HTTP заявка 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 за свободно изтегляне от адрес http://jakarta.apache.org/tomcat/. След като издърпаме последната версия, която представлява един **`.zip`** файл (примерно **`jakarta-tomcat-5.0.19.zip`**), трябва да разархивираме този файл 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 досадни проблеми.

Ако нямаме инсталиран JDK на нашия компютър, трябва да си инсталираме първо него. Можем да си го издърпаме от http://java.sun.com/j2se/.

Стартиране на Tomcat

За стартиране на сървъра Tomcat е необходимо първо в променливите на средата да добавим променливата **`JAVA_HOME`** със стойност директорията където е инсталиран JDK. Това може да стане например от конзолата с командата "**`set JAVA_HOME=C:\j2sdk1.4.2_01`**" или от настройките за променливите на средата (environment variables) на операционната система.

За да стартираме самия сървър Tomcat, трябва да изпълним скрипта за стартиране, който се намира в поддиректорията **`bin`** на основната директория на Tomcat сървъра. Например ако сме инсталирали Tomcat в директория **`C:\jakarta-tomcat-5.0.19`**, нашата **bin** директория ще бъде **`C:\jakarta-tomcat-5.0.19\bin`**.

В тази поддиректория **`bin`** има файл за стартиране **`startup.bat`** (**`startup.sh`** за Unix/Linux), с който можем да стартираме сървъра. Стартираме този файл. Появява се цяла поредица от съобщения, възникнали при началната инициализация на сървъра. Едно от последните съобщения казва, че сървърът слуша на порт **`8080`** за идващи HTTP заявки. Ако няма съобщения за грешки, то сървърът е стартирал успешно.

За да проверим дали всички работи нормално, можем да стартираме нашия Web-браузър и да въведем адреса http://localhost:8080/ (по подразбиране Tomcat работи на порт 8080, а не на стандартния за протокола HTTP порт 80). Ако всичко е наред, ще се появи заглавната страница на Tomcat. Можем да тестваме и стандартните примерни сървлети и JSP страници, като следваме връзките от главната страница.

Как да стартираме нашия сървлет

Да се върнем сега на проблема с изпълнението на примерния сървлет, който написахме преди малко. До момента знаем как се инсталира Tomcat и как се стартира. Остава да се научим как да deploy-ваме сървлети и как да ги извикваме от нашия Web-браузър. За целта трябва да направим ново Web-приложение в сървъра Tomcat, да копираме сървлета в него и да създадем специален конфигурационен файл, в който да опишем сървлета.

Създаване на ново Web-приложение

Освен поддиректорията **`bin`**, в директорията на Tomcat има още една важна поддиректория – **`webapps`**. В тази поддиректория стоят всички инсталирани Web-приложения. Например директорията **`webapps\ROOT`** е главната виртуална директория на сървъра и тя също представлява Web-приложение.

За нашите тестови цели трябва да създадем ново Web-приложение на сървъра (нова поддиректория в **`webapps`**).

Една добра стратегия при разучаване на някакъв непознат сървър е когато задавате имена на нови файлове, 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`**, **`nakov_cluster`** или нещо подобно, което ми подсказва че това име е избрано от мен и какво точно се крие зад това име – директория, услуга, клъстер или нещо друго.

Нека наречем нашето ново Web-приложение **nakov-webapp**. Създаваме директорията **nakov-webapp** в **webapps** поддиректорията на сървъра. Ако сега стартираме сървъра, всичко, което се намира в тази директория **nakov-webapp**, ще е достъпно от адрес http://localhost:8080/nakov-webapp/.

При стартиране Tomcat автоматично прочита всички физически поддиректории от поддиректорията си **webapps** и ги прави достъпни като виртуални директории в своя Web-сървър. Виртуална директория за даден Web-сървър наричаме ресурс, който се вижда през Web като директория в рамките на URL адреса, съответстващ този на Web-сървър. Например адресът http://localhost:8080/ съответства на главната виртуална директория за Web-сървъра работещ на хоста **localhost** на порт **8080**, а http://localhost:8080/nakov-webapp/ е адреса, който съответства на виртуалната директория **/nakov-webapp** на същия Web-сървър.

Копиране на файловете на сървлета

За да изпълним нашия сървлет е необходимо да го вкараме в новосъздаденото Web-приложение, т.е. някъде в неговата директория **nakov-webapp**. Съгласно стандартите от J2EE за Web-приложения трябва да създадем в **nakov-webapp** поддиректория с име **WEB-INF** и в нея още една поддиректория с име **classes** и да копираме компилирания сървлет в нея.

В нашия случай трябва да създадем директорията **C:\jakarta-tomcat-5.0.19\webapps\nakov-webapp\WEB-INF\classes** и в нея да копираме компилирания сървлет **FirstServlet.class**.

Конфигуриране на Web-приложението и описване на сървлета

Нашият компилиран сървлет се намира в директорията, в която стоят всички **.class** файлове, които са част от приложението. Понеже сървлетът също е **.class** файл, той си е на мястото. Остава да го регистрираме в Web-приложението, за да стане достъпен от виртуалната директория на приложението.

Всяко Java-базирано Web-приложение има конфигурационен файл с име **web.xml**, който се намира в поддиректория **WEB-INF** на директорията на приложението. В този файл се описват различни настройки на приложението, като име и описание на приложението, описание на използваните сървлети, описание на използваните потребителски тагове

(custom tags), описание на параметрите към приложението, описание на сървлет-филтри, настройки за сигурността и др.

За да направим нашия сървлет видим през URL адреса на нашето Web-приложение, трябва да го опишем във файла **web.xml**. Ето един пример как може да стане това:

web.xml

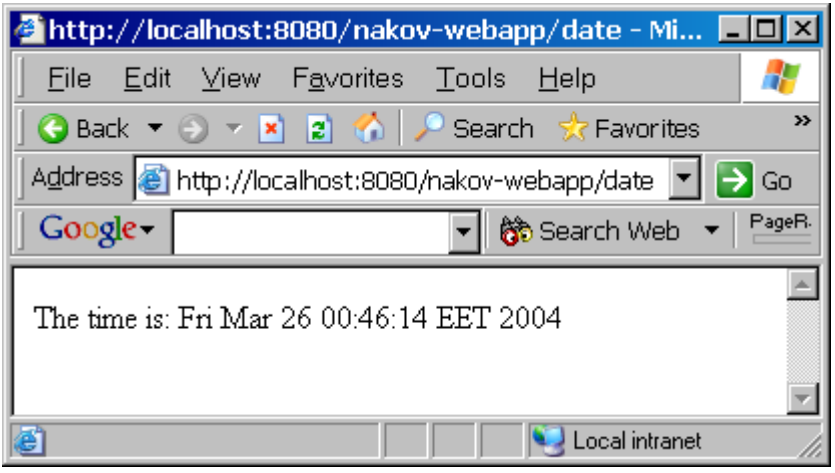
```
<web-app>
    <servlet>
        <servlet-name>dateServlet</servlet-name>
        <servlet-class>FirstServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>dateServlet</servlet-name>
        <url-pattern>/date</url-pattern>
    </servlet-mapping>
</web-app>
```

В тага **<servlet>** описваме съответствието между име на сървлет и име на клас, който стои за това име. Посоченият клас трябва да се намира в поддиректорията **WEB-INF\classes** на приложението. В тага **<servlet-mapping>** описваме съответствието между име на сървлет, което е дефинирано преди това и име на ресурс, под което ще е достъпен сървлета спрямо началния URL адрес на Web-приложението.

В нашия случай след като копираме този файл под име **web.xml** в поддиректорията **WEB-INF** на Web-приложението и рестартираме Tomcat сървъра, нашият сървлет става достъпен от адрес http://localhost:8080/nakov-webapp/date.

Примерният сървлет в действие

Можем да извикаме сървлета от стандартен Web-браузър като напишем URL адреса, към който той е закачен. Ето как изглежда резултатът от изпълнението на нашия сървлет в браузъра Internet Explorer:



Ето какъв е HTML кода, върнат от сървлета:

http://localhost:8080/nakov-webapp/date

```
<HTML>
The time is: Fri Mar 26 00:46:14 EET 2004
</HTML>
```

Как се извиква примерният сървлет

За да си изясним в по-голяма дълбочина какво се случва от момента, в който напишем URL адреса http://localhost:8080/nakov-webapp/date в Web-браузъра до момента, в който браузърът покаже резултата, ще проследим HTTP заявката и съответния HTTP отговор, които браузъра Internet Explorer 6.0 сървъра Tomcat 5.0 си обменят.

Когато потребителят въведе URL адреса на търсения ресурс, Web-браузърът намира по името на хоста (в случая **`localhost`**) IP адреса на Web-сървъра, от който трябва да поиска ресурса (в случая това в IP адрес **`127.0.0.1`**).

След като знае IP адреса на Web-сървъра, браузърът взима от URL адреса номера на порта, а ако не е указан порт използва порт **`80`**. В нашия случай се взима номер на порт **`8080`**. Браузърът отваря връзка по протокол TCP към намерения преди това IP адрес на търсения хост на извлечения от URL адреса порт.

След успешно свързване към Web-сървъра браузърът изпраща HTTP заявка за извличане на поискания ресурс. В нашия случай Internet Explorer 6.0 изпраща на Tomcat сървъра следното:

Internet Explorer → Tomcat

```
GET /nakov-webapp/date HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-shockwave-flash, */*
Accept-Language: bg
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
Q312461)
Host: nakov:8080
Connection: Keep-Alive
```

В заявката си Internet Explorer иска от Web-сървъра ресурс с име **/nakov-webapp/date** по протокол **HTTP/1.1**. Браузърът изпраща допълнителна информация относно какви ресурси може да приема и обработва, задава предпочитания език, с който иска да работи, задава каква компресия на данните разбира, задава идентификация, с която обяснява какъв браузър и коя версия е той самия, задава името на хоста, в рамките на Web-сървъра, за който се отнася заявката и накрая обяснява, че предпочита TCP връзката да остане отворена още известно време след като ресурсът бъде върнат. Заявката завършва с празен ред.

От гледна точка на сокет програмирането Tomcat е обикновен многопотребителски TCP сървър, който слуша на порт **8080** и си говори с клиентските приложения по протокол **HTTP**.

Когато Tomcat получи HTTP заявката от Internet Explorer, той я анализира за да разбере какво иска клиента. От частта **/nakov-webapp** на искания ресурс Tomcat разбира, че заявката се отнася за Web-приложението с име **nakov-webapp**. След това Tomcat установява, че в рамките на това Web-приложение е поискан ресурса **/date**. Този ресурс би могъл да бъде или някакъв сървлет или просто статичен документ, намиращ се в директорията на Web-приложението.

След справка в таблицата с регистрираните сървлети, която Tomcat е създал след като е прочел и анализирал конфигурационния файл на Web-приложението **web.xml**, сървърът установява, че трябва да извика сървлета **FirstServlet**. Този сървлет съответства на клас файла FirstServlet.class, който се намира в директорията **C:\jakarta-tomcat-5.0.19\webapps\nakov-webapp\WEB-INF\classes** на машината, на която работи сървъра. Ако класът все още не е зареден в паметта на Java виртуалната машина от **class loader**-а на Tomcat, той се зарежда и се подготвя за изпълнение.

Междувременно Tomcat е установил, че заявката идва по метода **GET** и съответно извиква метода **doGet(…)** на класа **FirstServlet**. Методът **doGet(…)** се изпълнява успешно и връща някакъв поток от символи, който сървърът приема от сървлета и въз основа не него формира HTTP отговора на клиентската заявката. В крайна сметка Tomcat изпраща на Internet Explorer следния HTTP отговор:

Tomcat → Internet Explorer

```
HTTP/1.1 200 OK
Content-Length: 60
Date: Fri, 26 Mar 2004 10:06:28 GMT
Server: Apache-Coyote/1.1

<HTML>
The time is: Fri Mar 26 12:06:28 EET 2004
</HTML>
```

HTTP отговорът започва с индикатора на протокол **HTTP/1.1**, код за успех **200** и текстово описание **OK**. Следва хедър поле, което указва дължината на върнатия ресурс, а след него стоят полетата за дата и за идентификация на сървъра. Забелязва се, че Tomcat се представя за сървър **Apache-Coyote/1.1**. Всъщност това не е измама. Наистина Tomcat 5.0 използва като подсистема, която обслужва HTTP протокола (HTTP listener) сървъра Apache Coyote, който е Java-базиран HTTP сървър с отворен код. Хедърите завършват с празен ред, а след него следва самия ресурс.

Най-накрая Web-браузърът получава отговора на HTTP заявката, която е изпратил и го интерпретира като HTML документ (понеже не е указано друго в хедърите на HTTP отговора) и го визуализира по подходящ начин.

Това е всичко, което се случва за едно най-обикновено извличане на ресурс през Web.

Особености при deploy-ването на Web-приложения

По подразбиране в Tomcat всички файлове от главната директория на едно приложение се публикуват във виртуална директория с име името на приложението. При някои версии на Tomcat (например 3.x и 4.x) всички **.class** файлове от **WEB-INF\classes** директорията автоматично се публикуват като сървлети във виртуална директория с име **<име_на_приложението>/servlet**. На това, обаче не може да се разчита и най-правилният начин да се публикува сървлет е да се опише в конфигурационния файл **web.xml**.

Специалната поддиректория **`WEB-INF`**, заедно с всички нейни поддиректории остава недостъпна през Web, въпреки че е поддиректория на Web-приложението.

Трябва да имаме предвид, че ако променяме файловете от директорията на нашето Web-приложение, промените ще са видими веднага, но ако променим и прекомпилираме някой файл от **`classes`** директорията (например нашият тестов сървлет), е необходимо да рестартираме Tomcat, за да влязат в сила промените. Това се дължи на с кеширането на заредените във виртуалната машина класове, което Tomcat прави за да постигне по-голяма производителност.

3.6. HTML форми и извличане на данните от тях

Всички сме използвали машини за търсене в Интернет като например Google и AltaVista и знаем, че те представляват Web-приложения, които приемат от потребителя няколко ключови думи и му връщат индексираните от търсачката страници, в които тези думи се срещат, сортирани по някакви критерии. 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 други символи. Например ако в Google зададем търсене на фразата **"Svetlin Nakov"**, ще получим URL много подобно на това: http://www.google.com/search?q=%22Svetlin+Nakov%22&ie=UTF-8&oe=UTF-8&hl=bg. Дясната част от URL-то след въпросителния знак съдържа данните, изпратени като параметри към това URL, кодирани по специален начин, наречен URL-encoding. Извличането на изпратени по този начин данни е важно за Web-приложенията, защото им позволява да приемат данни, въведени от потребителя в неговия Web-браузър.

Какво са HTML формите

Когато е бил измислен езикът HTML, той е бил предназначен за описание на форматирана текстова информация и не е имал тагове за вмъкване на контроли за въвеждане на данни. По-късно, когато Web-програмирането навлиза, е била осъзната нуждата потребителите от своя Web-браузър да могат да въвеждат и изпращат данни към сървъра и към езика са добавени HTML формите.

HTML формите позволяват в стандартен HTML документ да се вграждат полета за въвеждане на текст, контроли за избор на елемент от списък, контроли за избор в стил „да/не" и други контроли, чрез които потребителите могат да въвеждат данни и да ги изпращат на Web-приложението. В един HTML документ може да има няколко HTML форми, като всяка от тях може да има по няколко полета за въвеждане на данни. Всяка HTML форма си има действие – URL, към който се изпращат данните при изпращане (submit) на формата. Всяко поле от дадена HTML форма си има име, по което Web-приложението, което обработва получените данни, разбира кои данни в кое поле са били въведени.

Пример за HTML форма

Ето един пример за проста HTML форма с едно текстово поле, в което се очаква потребителят да въведе своето име:

HelloForm.html

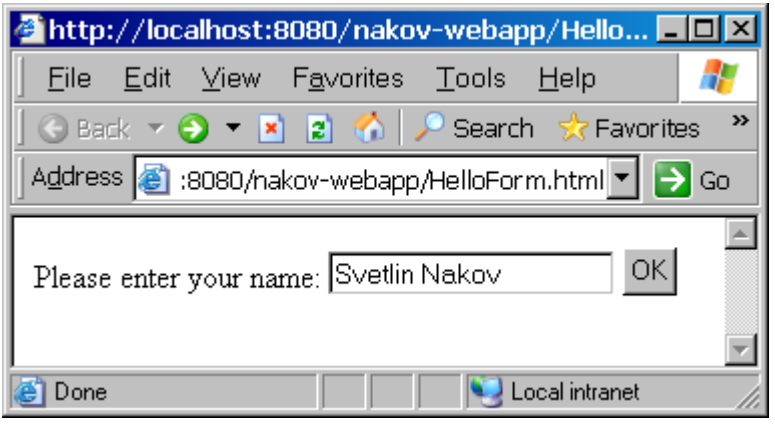
```
<html><body>
<form method="GET" action="HelloServlet">
    Please enter your name:
    <input type="text" name="user_name">
    <input type="submit" value="OK">
</form>
</body></html>
```

Тагът **<form>** има два атрибута. Атрибутът **method** задава дали ще се използва HTTP GET или HTTP POST заявка, а тагът **action** задава къде ще бъдат изпращани данните, когато потребителят ги попълни и реши да ги изпрати. Тагът **<input>** с атрибут **type="text"** задава текстово поле за въвеждане на данни, а същият таг с атрибут **type="text"** задава бутон за изпращане данните от формата. Атрибутът **name** е много важен, защото задава името на полето, по което Web-приложението ще разбере кои данни в кое поле са били въведени.

Тази форма е предназначена да извика сървлета **HelloServlet**, като му подаде стойността въведена от потребителя като параметър с име **user_name**.

Как изглежда примерната HTML форма

Ако визуализираме примерната HTML форма в стандартен Web-браузър, ще получим следния резултат:



Ако потребителят въведе стойност **Svetlin Nakov** в полето за име и натисне бутона за изпращане на формата, браузърът ще извлече параметрите и техните стойности от формата и ще ги изпрати към

указания с атрибута **action** обработчик (**HelloServlet**) под формата на заявка за извличане на URL адрес http://localhost:8080/nakov-webapp/HelloServlet?user_name=Svetlin+Nakov.

Java сървлети и извличане на данни, изпратени от клиента

При изпращане на HTML форма данните, въведени в нея, се изпращат към сървъра като част от HTTP GET или POST заявката. Java сървлетите имат вградена стандартна възможност за извличане на тези 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 се извършва от сървлет-контейнера 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).

Примерен сървлет за обработка на данни от HTML форма

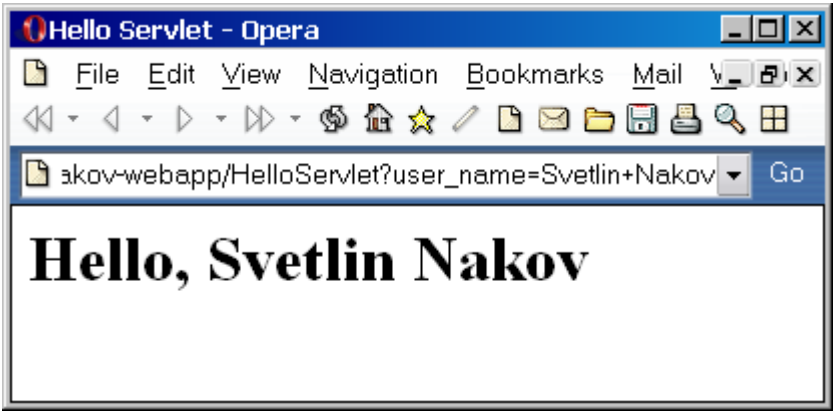
Ще разгледаме един много прост пример – сървлет, който получава като вход име на потребител (параметър с име **user_name**) и му казва "здравей":

HelloServlet.java

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

public class HelloServlet extends HttpServlet {
    public void doGet (HttpServletRequest aRequest,
        HttpServletResponse aResponse)
    throws ServletException, IOException {
        aResponse.setContentType("text/html");
        ServletOutputStream out = aResponse.getOutputStream();
        String userName = aRequest.getParameter("user_name");
        out.println("<html>");
        out.println("<head>");
        out.println("\t<title>Hello Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("\t<h1>Hello, " + userName + "</h1>");
        out.println("</body></html>");
    }
}
```

Сървлетът за обработка на данни от HTML форма в действие

Ако компилираме, deploy-нем и извикаме сървлета от Web-браузъра Opera със стойност „**`Svetlin Nakov`**" за параметъра **`user_name`**, ще получим резултат, подобен на следния:



Както се вижда от полето за адрес на браузъра, за да подадем стойност **`Svetlin Nakov`** за параметъра **`user_name`** на сървлета **`HelloServlet`**, е необходимо да извикаме URL адреса: http://localhost:8080/nakov-webapp/HelloServlet?user_name=Svetlin+Nakov. Символът *въпросителен знак* се използва за отделяне на ресурса от параметрите, които му се подават, а символът *интервал* се заменя със символа *знак за събиране*.

Използване на HTML форма

Разбира се, не е необходимо да пишем ръчно параметрите и техните стойности, когато искаме да подадем данни на даден сървлет. Вместо това можем да използваме HTML форми. Например формата от примера **`HelloForm.html`** може успешно да се използва за извикване на примерния сървлет.

Повече за HTML формите

Данните от HTML форма могат да бъдат предадени към сървъра по два различни начина – с GET или POST заявка. При GET HTTP заявки всички параметри се предават след URL адреса 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.

HTML автоматизират процеса на предаване на параметри между потребителския Web-браузър към сървърски скриптове, които

обработват тези параметри. Името на скрипта, който браузърът извиква при submit-ване на една HTML форма, се задава в атрибута **action**, а методът на HTTP заявката (GET или POST) се задава в атрибута **method** на тага **<form>**. Във формата могат да се включват 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 Web-браузърът се грижи да ги кодира по стандарта "URL-encoding" и да ги изпрати през URL-то или като част от заявката в зависимост дали методът за изпращане на формата е GET или POST. При използването на GET метод всички параметри се долепват към URL адреса, поради което обемът им не може да бъде много голям. При използването на POST метод всички параметри се предават скрито, отделно от URL адреса и потребителят вижда само името на скрипта, който е обработил данните, но не и самите данни.

Повече за сървлетите и извличането на параметри

Кой от двата метода GET или POST да се използва е въпрос на преценка от страна на Web-разработчика, но при всички случаи не се препоръчва използване на GET метода, ако се предават големи обеми от данни. При използване на метод GET сървлетът трябва да имплементира **doGet(…)**, а при използване на метод POST – **doPost(…)**.

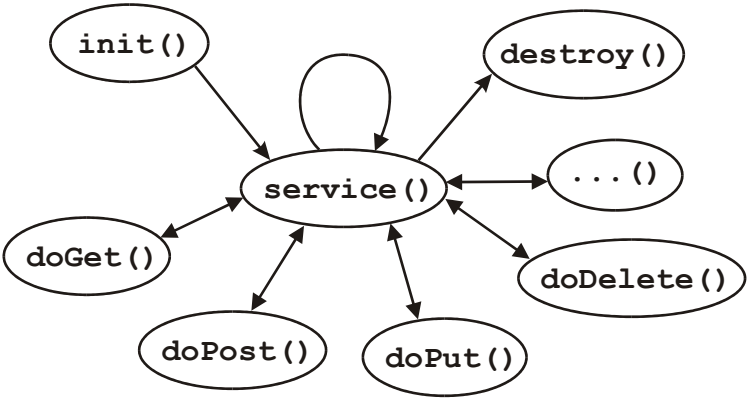
Извличането на стойностите на параметрите, подадени с GET или POST заявка става, както вече знаем, с метода **getParameter(…)**, който по име на параметър връща **String**. Ако параметър с посоченото име не е изпратен, методът връща стойност **null**, а ако параметърът е изпратен, но не му е зададена стойност, се връща празен низ.

Ако не знаем имената на очакваните параметри, можем да използваме метода **getParameterNames()** на класа **HttpServletRequest**, с който можем да получим имената на всички изпратени параметри.

Трябва винаги да внимаваме при задаването на имена на параметрите, защото малките и главните букви се различават.

3.7. Жизнен цикъл на сървлетите

Жизненият цикъл на сървлетите описва тяхното състояние 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:



Инициализация на сървлет

При първо извикване на сървлета Web-контейнерът извиква метода **`init(…)`**, дефиниран в класа **`HttpServlet`**. 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-късно.

Обслужване на заявки

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 метода на заявката и в зависимост от това дали той е GET, POST, PUT, DELETE или друга команда, извиква съответно един от методите **`doGet(…)`**, **`doPost(…)`**, **`doPut(…)`**, **`doDelete(…)`** или съответно друг.

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 **<form 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 препоръчително да припокриваме директно метода **service(…)**, защото той се грижи за правилната обработка на HEAD заявките и за още други важни неща. Вместо това е по-добре да имплементиране обработката на данните в метода **doGet(…)**, а от **doPost(…)** просто да извикваме **doGet(…)** с подадените параметри. Това е препоръчителният начин за обработка на данни, които се очаква да пристигат както чрез GET, така и чрез POST заявки.

Моделът на инстанциране и обслужване на конкурентни 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 отново от **.class** файла. В рамките на едно Web-приложение един сървлет се инстанцира само веднъж, т.е. сървърът създава само най-много по една инстанция от всеки един сървлет.

Когато няколко клиента поискат даден сървлет едновременно, Web-контейнерът стартира едновременно няколко нишки (threads) и изпълнява във всяка от тях сървлета. Понеже всеки сървлет има само една инстанция на сървъра, то класът, който реализира даден 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. Заради тази особеност 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 други общи ресурси, които сървлетът използва.

Повечето сървлет-контейнери използват модела “thread pool” за управление на нишките, които обработват клиентските заявки. При този модел винаги има определен брой нишки, създадени предварително, които стоят и чакат да им бъде зададена заявка за изпълнение. При

извикване на някой сървлет или JSP страница от thread pool-а се изважда свободна нишка и тя се използва за изпълнение на заявката. След това, когато изпълнението на заявката приключи, нишката става отново свободна и се връща обратно в thread pool-а. Така се постига по-голяма производителност, защото се спестява времето за създаване и унищожаване на нишки при всяка заявка.

Унищожаване на сървлет

Когато Web-контейнерът по някаква причина реши да премахне от паметта един сървлет (например при намеса на администратора), се извиква методът **`destroy()`** на класа **`HttpServlet`**. В реализацията на този метод сървлетите трябва да освободят заетите от тях ресурси и да финализират работата си.

Методът **`destroy()`** се използва по-рядко от метода **`init()`**, защото Java освобождава автоматично някои типове ресурси, като например използваната памет и по-рядко се налага ръчно да се освобождават ресурси в **`destroy()`** метода.

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 по съвсем друг модел, обикновено от специална компонента на системата известна като “connection pool”.

Сървлет за броене на посетителите

С настоящия пример ще илюстрираме две неща – използване на методите от жизнения цикъл на сървлетите и динамично генериране на графични изображения от сървлет. Нашият сървлет ще представлява класически брояч на посетителите на даден Web-сайт (web counter), който визуализира текущата си стойност във вид на JPEG изображение. Ето една примерна реализация:

ImageCounterServlet.java

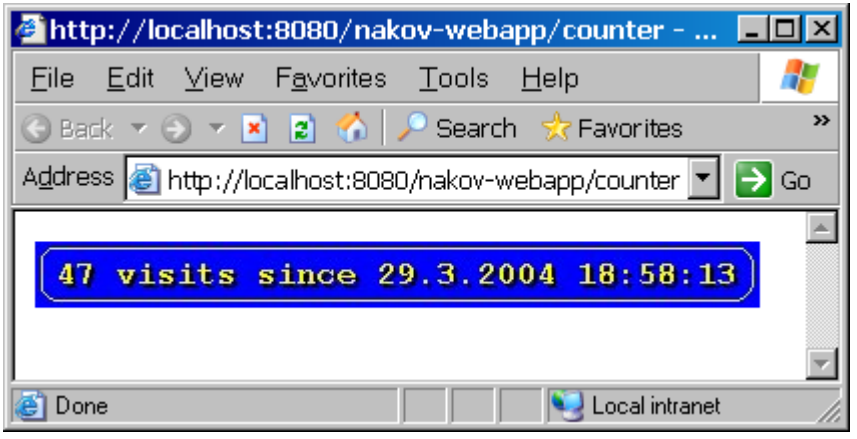
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.*;
import java.awt.image.*;
import com.sun.image.codec.jpeg.*;
import java.io.*;
import java.util.Date;
import java.text.SimpleDateFormat;

public class ImageCounterServlet extends HttpServlet {
```

```
    private static final float JPEG_QUALITY = (float) 0.85;

    private String mStartDate;
    private int mVisitCounter;

    /**
     * Called by the servlet container when the servlet is
     * instantiated. Initializes the counter and start date.
     */
    public void init() {
        Date now = new Date();
        SimpleDateFormat dateFormatter =
            new SimpleDateFormat("d.M.yyyy HH:mm:ss");
        mStartDate = dateFormatter.format(now);
        mVisitCounter = 0;
    }

    /**
     * Creates a graphical image and draws given text with
     * with "Monospaced" yellow font on a red background.
     */
    public BufferedImage createImage(String aMsg) {
        Font font = new Font("Monospaced", Font.BOLD, 16);
        FontMetrics fm = new Canvas().getFontMetrics(font);
        int width = fm.stringWidth(aMsg) + 22;
        int height = fm.getHeight() + 11;
        BufferedImage image = new BufferedImage(
            width, height, BufferedImage.TYPE_INT_RGB);
        Graphics g = image.getGraphics();
        g.setColor(Color.blue);
        g.fillRect(0, 0, width, height);
        g.setColor(Color.yellow);
        g.drawRoundRect(3, 3, width-7, height-7, 15, 15);
        g.setFont(font);
        g.setColor(Color.black);
        g.drawString(aMsg, 11+2, 4 + fm.getAscent()+2);
        g.setColor(Color.yellow);
        g.drawString(aMsg, 11, 4 + fm.getAscent());
        return image;
    }

    /**
     * Called by the servlet container on HTTP GET request.
     * Increases the counter and sends as a response a JPEG
     * image that contains the counter value along with some
     * additional text.
     */
    public void doGet(HttpServletRequest aRequest,
        HttpServletResponse aResponse)
    throws IOException, ServletException {
        String msg;
        synchronized(this) {
```

```
            mVisitCounter++;
            msg = "" + mVisitCounter + " visits since " +
                mStartDate;
        }
        BufferedImage image = createImage(msg);
        aResponse.setContentType("image/jpeg");
        OutputStream out = aResponse.getOutputStream();
        JPEGImageEncoder encoder =
            JPEGCodec.createJPEGEncoder(out);
        JPEGEncodeParam jpegParams =
            encoder.getDefaultJPEGEncodeParam(image);
        jpegParams.setQuality(JPEG_QUALITY, false);
        encoder.setJPEGEncodeParam(jpegParams);
        encoder.encode(image);
        out.close();
    }

    /**
     * Called by the servlet container on HTTP POST request.
     * Just delegate to doGet() method.
     */
    public void doPost(HttpServletRequest aRequest,
        HttpServletResponse aResponse)
    throws IOException, ServletException {
        doGet(aRequest, aResponse);
    }
}
```

Сървлетът за броене на посетители в действие

След компилиране и deploy-ване под име **`counter`** в рамките на приложението **`nakov-webapp`** и извикване от Internet Explorer сървлетът връща резултат, подобен на следния:



При вглеждане в изображението може да се забележи дори загубата на качество, което се е получило при компресията на динамично генерираната картинка в JPEG формат. При първото извличане на сървлета, той започва броенето на посетителите, а при всяко следващо извикване увеличава брояча с 1.

Как работи сървлетът за броене на посетители

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 колко пъти, които е бил извикван чрез GET или POST заявка.

При извикване на **`doGet(…)`** метода, сървлетът увеличава с 1 брояча на посетителите и генерира текстово съобщение, което обяснява колко пъти е била посетена страницата от първото извикване на сървлета. Заради възможността няколко потребители едновременно да поискат страницата, достъпът до член-променливите на сървлета работи синхронизирано. 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 на сървлета, който както знаем е един и същ за цялото Web-приложение, независимо от коя нишка се използва.

Понеже 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 конкурентно извикване от много потребители едновременно.

За динамичното генериране на изображение по зададен текст в метода **`createImage(…)`** се използват стандартните средства от библиотеката AWT. Първо се изчисляват размерите на текста при шрифта, който ще бъде използван и се създава изображение от класа **`BufferedImage`** с малко по-големи размери, в което ще бъде изобразен текста. От вече създадения **`BufferedImage`** обект се взима неговия обект за графична манипулация **`Graphics`** и чрез поредица от прости графични операции се изобразява правоъгълник със заоблени ъгли и в него се отпечатва зададения текст. Първо изображението се запълва със син цвят, след това се печата текста с черен цвят, а след него се печата с жълто същия текст, но отместен малко по-вляво и по-нагоре. Така се получава усещането за сянка на текста.

В **`doGet(…)`** метода, след като изображението е вече 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 ниво на качеството 85%. Полученото JPEG изображение се изпраща на клиента като изход от сървлета.

За да може Web-браузърът да разпознае изпратения поток от данни като JPEG картинка, а не като текст или нещо друго, в header-а на HTTP отговора на заявката се добавя един ред със съдържание "**`Content-type: image/jpeg`**". По подразбиране, ако сървлетът не укаже друго, за тип на изпратените данни се приема форматът "**`text/html`**". Ако пропуснем да укажем, че сървлетът връща резултата си във формат „**`image/jpeg`**", Web-браузърът ще приеме, че изпратеното изображение е текст и ще визуализира на екрана някакви странна безсмислена последователност от символи („джуджуфлечки", „каракацили" или „маймуняци", както още ги наричат в различните диалекти).

Реализацията на метода **`doPost(…)`** просто извиква метода **`doGet(…)`** и това позволява на сървлета да отговаря по еднакъв начин както на GET, така и на POST заявки идващи от клиента.

3.8. Поддръжка на потребителски сесии

В тази тема ще разгледаме как Java-базираните Web-приложения могат да обслужват много потребители едновременно и независимо един от друг с помощта на т. нар. потребителски сесии.

Какво е потребителска сесия

Потребителска сесия наричаме периода, през който един потребител си взаимодейства с дадено Web-приложение – от подаването на първата заявка към това приложение до затварянето на Web-браузъра или изтичането на някакъв ограничителен период от време, през който потребителят не изпраща нито една заявка към приложението.

Проследяването на последователността от заявки, извършени от един потребител се нарича проследяване на неговата сесия. Ако двама потребителя работят едновременно с едно и също Web-приложение, те имат две различни сесии.

Пример за Web-приложение, което проследява потребителската сесия, е Web-базираната система за електронна поща Yahoo Mail. 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 сървъра с име mail.yahoo.com разпознава различните потребители и проследява техните сесии. Приложението получава постоянно голямо количество заявки от най-различни потребители. В зависимост от това кой потребител е дал HTTP заявка към Web-приложението, то разпознава неговата сесия и дава достъп до неговите email-и, а не до email-ите на останалите потребители, работещи в същия момент.

Възможно е дори от един и същ компютър да се осъществят едновременно няколко независими една от друга 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 през двата браузъра като два различни потребителя. В рамките на браузъра Netscape, той ще има създадена една сесия с Web-приложението, 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 дължи на механизма, по който сървърът различава потребителите един от друг.

Начини за различаване на потребителите един от друг

Както 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 е изпратена. Затова Web-приложенията трябва да полагат допълнителни усилия за проследяване и разграничаване на потребителите един от друг.

Има два основни начина за проследяване на потребителската сесия – с **cookies** и с добавяне на допълнителен параметър към URL адреса.

Идентификация на сесията с cookies

Кукитата (cookies) позволяват 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 използва дълго време. При създаване на cookie му се задават параметри, в които се указва колко дълго да бъде да бъде съхранявано това cookie на машината на клиента.

Посредством cookies Web-приложенията могат да записват на машината на потребителя някакъв уникален идентификатор на сесия (session id) и след това да получават този идентификатор при всяка заявка и по него да разпознават отделните потребители.

Идентификация на сесията чрез параметри на HTTP заявките

Другият начин за следене на потребителските сесии е чрез добавяне на допълнителен параметър към URL адреса. При започване на работа за потребителя се генерира уникален идентификатор на сесия (session id) 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 за добавяне на параметри към всеки hyperlink и затова се използва рядко, най-вече когато клиентският браузър не поддържа cookies или потребителят ги е забранил. С cookies усилията за проследяване на потребителите са значително по-малки.

Потребителски сесии в Java

В Java-базираните Web-приложения поддръжката на потребителски сесии е напълно автоматична. Програмистът не е нужно да изпраща и чете cookies или да добавя допълнителни параметри към URL адреса и да ги разпознава след това. Достатъчно е да се използва стандартното API за работа със сесии, което се дава от спецификацията за Java Web-приложения. Най-важният клас, свързан с управлението на потребителските сесии е класът **`javax.servlet.http.HttpSession`**, който представя потребителската сесия. По време на изпълнението на

сървлет можем да получим обект, асоцииран с текущата сесия от **HttpRequest** обекта по следния начин:

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

При извикване на **getSession()** метода ако сесия за текущия потребител все още не съществува, такава ще бъде създадена. Взимането на **HttpSession** обекта за един и същ потребител в рамките на едно Web-приложение 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 **HttpSession** обекта посредством методите **setAttribute(key, value)** и **getAttribute(key)** могат да бъдат съхранявани и извличани по-късно произволни данни за потребителя (на практика всякакви Java обекти). След като веднъж са съхранени в сесията, тези обекти остават достъпни по време на всяка заявка от съответния потребител в рамките на неговата сесия.

Пример за използване на потребителски сесии в Java

Ще илюстрираме използването на потребителските сесии с един пример, който се състои от два сървлета. Единият сървлет служи за автентикация на потребителите, а другият предоставя различна информация в зависимост от името на текущия потребител, което е било използвано при автентикацията. Ето как изглежда сървлетът за автентикация на потребителите:

LoginServlet.java

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

public class LoginServlet extends HttpServlet {
    public void doGet(HttpServletRequest aRequest,
        HttpServletResponse aResponse)
    throws IOException, ServletException {
        String username = aRequest.getParameter("username");
        String password = aRequest.getParameter("password");
        PrintWriter out = aResponse.getWriter();
        if ((username==null) || (password==null)) {
            showLoginForm("Please login:", out);
        } else if (username.equals(password)) {
            HttpSession session = aRequest.getSession();
            session.setAttribute("USER", username);
            aResponse.sendRedirect("main");
        } else {
```

```
            showLoginForm("Invalid login! Try again:", out);
        }
    }

    public void doPost(HttpServletRequest aRequest,
        HttpServletResponse aResponse)
    throws IOException, ServletException {
        doGet(aRequest, aResponse);
    }

    private void showLoginForm(
            String aCaptionText, PrintWriter aOutput) {
        aOutput.println(
            "<html><title>Login</title><body>\n" +
            "<form method='POST' action='login'>\n" +
            aCaptionText + "<br>\n" +
            "<input type='text' name='username'><br>\n" +
            "<input type='password' name='password'><br>\n" +
            "<input type='submit' value='Login'>\n" +
            "</body></form></html>"
        );
    }
}
```

Ето как изглежда и другият сървлет, който използва данните от сесията, създадена от Login-сървлета:

MainServlet.java

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

public class MainServlet extends HttpServlet {
    public void doGet(HttpServletRequest aRequest,
        HttpServletResponse aResponse)
    throws IOException, ServletException {
        HttpSession session = aRequest.getSession();
        String username = (String) session.getAttribute("USER");
        PrintWriter out = aResponse.getWriter();
        if (username == null) {
            showMainForm("Not authenticated. Please " +
                "<a href='login'>login</a> first.", out);
        } else {
            showMainForm("Welcome, " + username + "!", out);
        }
    }

    private void showMainForm(String aText, PrintWriter aOut) {
        aOut.println("<html>" + aText + "</html>");
    }
}
```

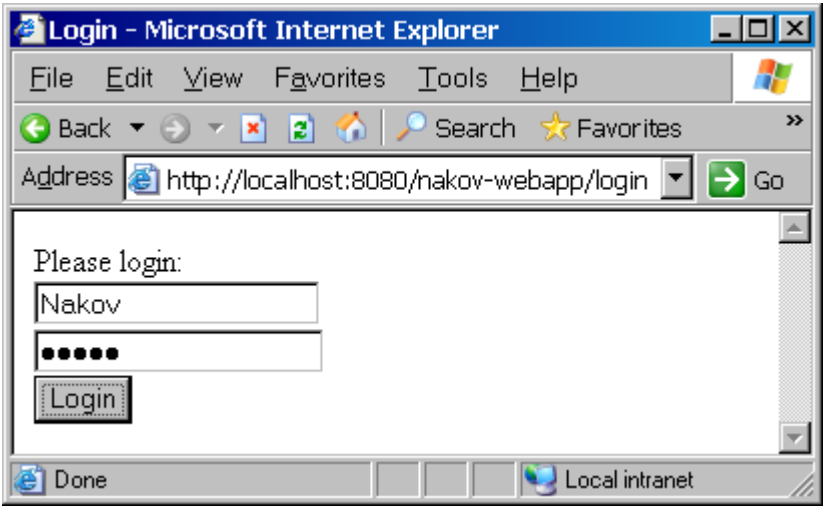
Примерът за използване на потребителски сесии в действие

Нека компилираме двата сървлета и ги deploy-нем на сървъра в рамките на Web-приложението **nakov-webapp** и ги конфигурираме така, че да са достъпни съответно с имена **login** и **main** от виртуалната директория на Web-приложението. За целта компилираме сорс кода на сървлетите, получаваме файловете **LoginServlet.class** и **MainServlet.class** и ги копираме в директорията, където се разполагат класовете на Web-приложението с име **nakov-webapp**. При нашата инсталация на Tomcat 5.0 тази директория е **C:\jakarta-tomcat-5.0.19\webapps\nakov-webapp\WEB-INF\classes**. След това редактираме конфигурационния файл **web.xml** на приложението, който се намира в поддиректорията **WEB-INF**, за да укажем под какви имена да са достъпни нашите сървлети:

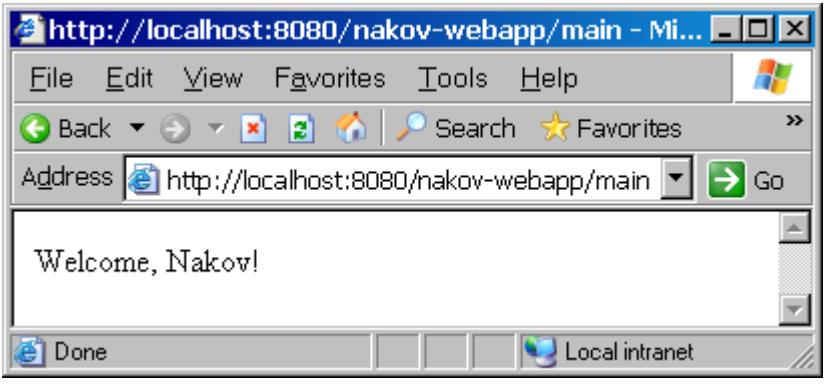
web.xml

```
<web-app>
    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>LoginServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>MainServlet</servlet-name>
        <servlet-class>MainServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/login</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>MainServlet</servlet-name>
        <url-pattern>/main</url-pattern>
    </servlet-mapping>
</web-app>
```

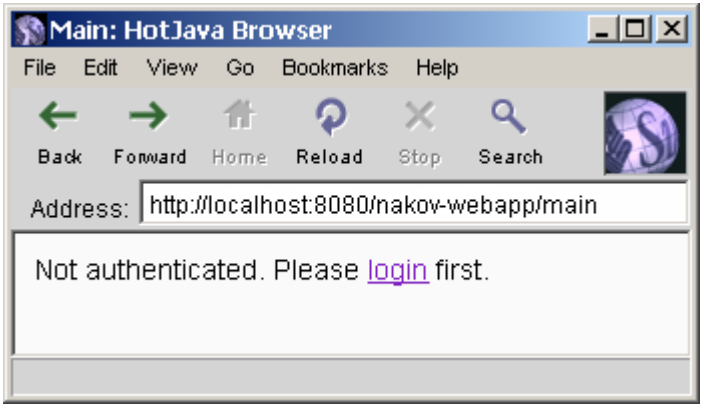
След като стартираме сървъра Tomcat и въведем в Internet Explorer адреса http://localhost:8080/nakov-webapp/login, ще получим резултата от изпълнението на сървлета **LoginServlet**:



Ако въведем в полето за име на потребител „**Nakov**" и в полето за парола отново „**Nakov**" и изпратим формата, ще бъдем пренасочени към адрес http://localhost:8080/nakov-webapp/main и сървлетът **MainServlet** ще ни поздрави по името, което сме въвели в предходната страница:



Ако в този момент някой друг потребител с друг Web-браузър се опита директно да отиде на страницата http://localhost:8080/nakov-webapp/main като пропусне автентикацията от **LoginServlet**-а, ще получи следния резултат:



Как работят сървлетите от примера

Сървлетът за автентикация **`LoginServlet`** при получаване на клиентска заявка по GET или POST метод изпълнява метода си **`doGet(…)`**. В него той проверява параметрите, с които е бил извикан. Ако не са му подадени параметрите потребител и парола, сървлетът връща HTML форма, в която потребителят да ги въведе. Тази HTML форма има зададен **`action="login"`**, което означава, че попълнените в нея данни се изпращат отново към същия **`LoginServlet`**. Указно е методът на формата да е **`POST`**, за да не остават името на потребителя и паролата му в URL адреса на Web-браузъра.

При извикване с някакви стойности на параметрите **`username`** и **`password`** сървлетът проверява това е валиден потребител. За простота в нашия пример валидни комбинации от потребителско име и парола са всички, в които потребителското име съвпада с паролата. Когато сървлетът разпознае валидна комбинация от потребителско име и парола, записва в автоматично създадената за текущия потребител сесия под ключ "**`USER`**" въведеното потребителско име. След това потребителят се препраща към основния сървлет **`MainServlet`** чрез метода **`sendRedirect(URL)`** на класа **`HttpServletResponse`**.

При невалидна комбинация от име на потребител и парола се връща отново HTML формата за въвеждане на тези данни, придружена от съобщение за неуспешна автентикация.

Основният сървлет извлича от **`HttpSession`** обекта стойността под ключ "**`USER`**" и по нея разбира дали текущият потребител е автентикиран. Ако потребителят е автентикиран, стойността от ключа

"**USER**" е неговото потребителско име, а в противен случай тази стойност е **null**. Ако текущият потребител е известен, му се изпраща кратък поздрав по име, а в противен случай му се изпраща съобщение, с което му се обяснява, че трябва първо да се идентифицира пред системата чрез login сървлета.

Още за управлението на потребителските сесии в Java

Както видяхме, проследяването на потребителската сесия се извършва автоматично от Web-контейнера и на разработчика се дава наготово достъп до **HttpSession** обекта, който се създава и поддържа автоматично за всеки клиент на Web-приложението. В този обект могат да се записват различни Java обекти под различни ключове и сървлет-контейнерът гарантира той е различен за различните потребители и е един и същ за последователните заявки на един и същ потребител.

По принцип една сесия е валидна известно време (например 3 минути), за което време ако не се използва, автоматично изтича (унищожава се). Чрез метода **setMaxInactiveInterval()** на класа **HttpSession** можем да задаваме точно колко да бъде времето на неактивност в милисекунди, за което една сесия изтича и се изтрива от сървъра. Изтичането на сесиите (session expiration) е полезно от съображения за сигурност. След като един потребител е автентикиран веднъж, е полезно неговата автентикация да 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 на потребителя (login) предлага и изход от системата (logout), което прекратява потребителската сесия. Реализацията на logout в нашия пример може да се реализира като се изтрие стойността с ключ "**USER**" от сесията чрез извикване на **removeAttribute("USER")**. За анулиране на сесията има и стандартен метод **invalidate()**, който изтрива всичката информация от нея.

Потребителските сесии и сигурността

Използването на потребителски сесии крие доста опасности за сигурността на Web-приложението. Обикновено за идентификатор на сесия (session id) се използва голямо случайно число, което не може лесно да бъде предсказано. Възможни са няколко атаки върху потребителската сесия:

- отгатване на идентификатора на сесията, например чрез brute-force атака или ако алгоритъмът за генериране на уникален session id връща предсказуем резултат;
- подслушване на идентификатора на сесията, например чрез подслушване на мрежовия трафик;
- откradване на идентификатора на сесията, например от полето **`HTTP REFERER`** от HTTP хедъра на GET или POST заявката, което съдържа URL адреса на последната посетена страница или чрез инжектиране на злонамерен код (cross-site scripting).

При отвличане на идентификатора на потребителската сесия (session hijacking) е възможно злонамерени потребители да осъществят достъп до чужди данни, принадлежащи на потребителя, на когото са отвлекли сесията.

За защита от атаки върху потребителската сесия се използва задължително SSL криптирана връзка, но само това в някои случаи не е достатъчно. Допълнително трябва да се комбинира идентификация с cookies и идентификация с параметри в URL адреса. Добро ръководство за осигуряване на сигурността на потребителската сесия има на адрес: http://www.technicalinfo.net/papers/WebBasedSessionManagement.html.

3.9. Java Server Pages (JSP)

До сега 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 едно и също Web-приложение като използват HTTP сесии. Предстои ни да навлезем в технологията Java Server Pages (JSP). Ще обясним основните тагове в JSP, ще изясним каква е връзката между JSP и сървлетите и ще покажем как чрез тази технология, усилията за разработка на Web-приложения значително се намаляват.

Какво е 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 извличане на този документ. Фрагментите програмен код, вградени в HTML документа се ограждат със специални тагове и се наричат **JSP скриптлети**. HTML документи, съдържащи скриптлети, 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 преди това всички JSP скриптлети, вградени в него, се изпълняват и се заместват с резултата от изпълнението им. Така един JSP документ, който е смесица от HTML и Java код, след изпълнението си се превръща в динамично генериран чист HTML документ.

Хубавото при тази технология е, че статичният HTML в един документ си остава статичен, а само на местата, където се налага да има динамични части, се използва код на Java. По този начин HTML документът остава четим и за Web-дизайнерите, които могат и да не разбират от Java. За разлика от сървлетите, при JSP-тата не е нужно всичкият HTML текст на генерирания документ да се отпечатва в изходния поток ред по ред чрез извиквания от вида `out.println(…)`. Вместо това може всички статични части да си останат статични, а всички динамични части да се напишат като JSP скриптлети. Това силно улеснява работата на разработчика, подобрява четимостта на кода и опростява поддръжката му.

JSP скриптлети

Скриптлетите в JSP-страниците се ограждат с таговете `<%` и `%>` съответно за начало и край. При извикване на JSP-страница кодът, ограден от тези тагове не се връща директно на клиента, а се изпълнява

по време на заявката и се връща резултатът от него. Ето един пример за JSP-страница, която отпечатва текущата дата и час, вграждайки в себе си скриптлет:

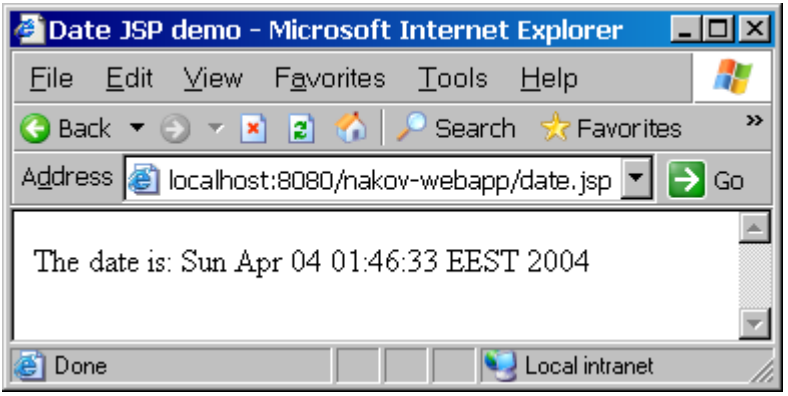
date.jsp

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

Примерната JSP-страничка много прилича на обикновен HTML документ, само че съдържа JSP скриптлет, който при извикване отпечатва текущата дата и час чрез стандартните средства на Java.

Как да изпълним примерната JSP страничка

Стартирането на JSP страничка е значително по-просто отколкото стартирането на сървлет. Всичко, което трябва да направим за да изпълним примерната страничка, е да изкопираме файла **date.jsp** в директорията на Web-приложението, да стартираме сървлет-контейнера и да извикаме тази страничка от виртуалната директория на Web-приложението чрез стандартен Web-браузър. Например можем да копираме **date.jsp** в поддиректорията **nakov-webapp** на директория **webapps** от Tomcat сървъра, да стартираме Tomcat и да поискаме URL адреса http://localhost:8080/nakov-webapp/date.jsp. Резултатът е същия, който се получи при изпълнението на сървлета **DateServlet**:



Не е необходимо да пишем нищо в конфигурационния файл на Web-

приложението **web.xml**. JSP страниците не е нужно да се конфигурират.

При Tomcat 5.0 и следващите версии има особеност – задължително е в директорията на приложението да съществува поддиректорията **WEB-INF**, защото иначе Tomcat не разпознава приложението като валидно J2EE Web-приложение и дава съобщение, че поисканият файл не е намерен. Ако искаме да тестваме дадено JSP, е най-добре да го копираме в директорията на някое вече работещо Web-приложение, което си има директория **WEB-INF** и конфигурационен файл **web.xml**.

Стандартни обекти в JSP

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

request – за достъп до HTTP заявката и параметрите, които клиентът е изпратил към нея

response – за управление на отговора на HTTP заявката

out – изходен текстов поток за отговора на HTTP заявката

session – за управление на потребителските сесии

application – за достъп до данните, съхранявани в контекста на Web-приложението

За удобство на програмиста тези обекти са достъпни от всички скриптлети в JSP-страницата. В нашия пример използвахме обекта **out**, чрез който отпечатахме текущата дата в изходния поток на JSP-страницата. В други случаи ще използваме и другите обекти. Целта е на автоматично създадените обекти е да се намали обемът на кода, който програмистът механично пише при работа със сървлети.

Технологията Java Server Pages предоставя на Web-разработчика освен скриптлети и други тагове. Ще 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. JSP-изразите имат следния синтаксис:

```
<%= израз %>
```

и всеки такъв израз е еквивалентен на скриптлета

```
<% out.write(израз) %>
```

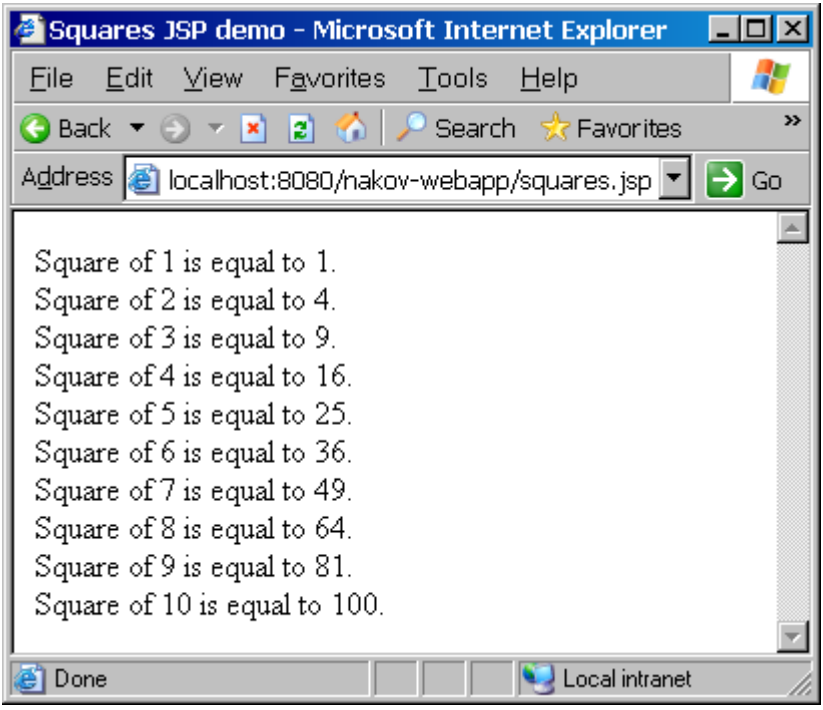
Като пример за използване на JSP-изрази можем да дадем следната JSP-страница, която отпечатва числата от 1 до 10 и техните квадрати:

squares.jsp

```
<html>
    <head><title>Squares JSP demo</title></head>
    <body>
        <% for (int i=1; i<=10; i++) { %>
            Square of <%= i %> is equal to <%= i*i %>.
            <br />
        <% } %>
    </body>
</html>
```

Забележете, че когато чрез JSP-скриптлети 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 – използване на чист HTML в тялото на цикли и if-конструкции. Както се вижда, възможно е един цикъл да започва в един скриптлет и да завършва в друг, а тялото му да е чист HTML, разположен между двата скриптлета. Това е естествено следствие от начина, по който JSP документите се трансформират в сървлети или по-точно в Java copc-код на сървлети, който след това се компилира до класове.

Ето как изглежда резултатът от изпълнението на **squares.jsp**:



За разлика от сървлетите, за изпълнението на JSP страница не е необходимо да спираме и след това да стартираме отново сървъра. Достатъчно е да копираме новия JSP файл в директорията на Web-приложението и той веднага става видим от виртуалната директория на приложението. Можем да правим дори нещо повече: при пуснат сървър можем да променяме JSP файловете от приложението и при следващото им извикване сървърът ще разбере за промяната и ще използва обновената версия.

Как JSP страниците се преобразуват до сървлети

JSP страниците представляват файлове в Web-приложението, които при първо извикване се трансформират от Web-контейнера в Java сървлети и след това се изпълняват като всички останали сървлети. Поради тази причина всичко, което знаем за сървлетите, важи и за JSP страниците.

Можем да приемем, че когато Web-контейнерът трансформира един JSP документ в Java сървлет, той замества всички редове, представляващи чист HTML текст, с програмен код, който отпечатва този текст в изходния поток на отговора на HTTP заявката. Заедно с това всички JSP

скриптлети, които съдържат Java код, си остават същите, а специалните JSP тагове се преобразуват в код, който се записва на различни места в сървлета в зависимост от самите тагове. Например, ако първият ред от нашето JSP, съдържа статичния текст

```
<html>
```

той ще се бъде заменен при трансформацията с програмния код

```
out.write("<html>\n");
```

След множество преобразувания полученият сорс-код се записва в тялото на един метод, който се извиква от метода **service(…)** на резултатния сървлет и се изпълнява при клиентска HTTP заявка. JSP страниците обслужват по еднакъв начин както GET, така и POST заявки.

Ето един пример как Web-контейнерът Tomcat 5.0.19 компилира JSP страничката **squares.jsp** до сорс-код на сървлет. Полученият Java клас не е точно Java сървлет, но много прилича на такъв:

squares_jsp.java

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class squares_jsp
    extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    private static java.util.Vector _jspx_dependants;

    public java.util.List getDependants() {
        return _jspx_dependants;
    }

    public void _jspService(HttpServletRequest request,
            HttpServletResponse response)
            throws java.io.IOException, ServletException {
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;
```

```
        try {
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html");
            pageContext = _jspxFactory.getPageContext(this,
                request, response, null, true, 8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();
            _jspx_out = out;

            out.write("<html>\r\n");
            out.write("\t<head><title>Squares JSP demo" +
                "</title></head>\r\n");
            out.write("\t<body>\r\n");
            out.write("\t\t");
            for (int i = 1; i <= 10; i++) {
                out.write("\r\n");
                out.write("\t\t\tSquare of ");
                out.print(i);
                out.write(" is equal to ");
                out.print(i * i);
                out.write(".\r\n");
                out.write("\t\t\t<br />\r\n");
                out.write("\t\t");
            }
            out.write("\r\n");
            out.write("\t</body>\r\n");
            out.write("</html>\r\n");
        } catch (Throwable t) {
            if (!(t instanceof SkipPageException)) {
                out = _jspx_out;
                if (out != null && out.getBufferSize() != 0)
                    out.clearBuffer();
                if (_jspx_page_context != null)
                    _jspx_page_context.handlePageException(t);
            }
        } finally {
            if (_jspxFactory != null)
                _jspxFactory.releasePageContext(
                    _jspx_page_context);
        }
    }
}
```

Кодът не е много лесен за четене, но това е все пак машинно-генериран код и по принцип не е предназначен да бъде четен от друг освен от компилатора. При внимателно разглеждане може да се забележи точно съответствие със сорс-кода на страничката **`squares.jsp`**.

При сървър Tomcat кодът, получен при компилирането на всички изпълнени от Web-контейнера JSP страници, се намира някъде из поддиректорията **work** на сървъра (за различните версии на Tomcat точното местоположение е различно).

Как се изпълнява JSP страница

При първо извикване на една JSP страница, Web-контейнерът я трансформира в Java сорс-код на сървлет, компилира го и получава **.class** файл. След това зарежда този клас в паметта и го изпълнява както изпълнява сървлети. При всяко следващо извикване, ако JSP-то не е променено, то не се компилира, а директно се изпълнява от заредения в паметта компилиран код, получен при първото извикване. Ако JSP-то е променено, то автоматично се прекомпилирва и тогава се изпълнява.

Едно от облекченията за разработчика при JSP технологията е, че не е необходимо да се рестартира сървърът при всяка промяна на едно JSP, за да се види резултата от нея, както трябва да се прави при промяната на сървлет при повечето Web-контейнери.

Още за JSP страниците

Понеже JSP-тата са сървлети, те имат същия жизнен цикъл, като сървлетите и могат да се възползват от всички техни предимства, свойства и особености. Например от JSP скриптлет чрез метода **getParameter(…)** на **HttpServletRequest** обекта (който е достъпен като локална променлива с име **request**) можем да взимаме и обработваме параметрите, подадени при HTTP заявката. Можем да управляваме потребителската сесия чрез **HttpSession** обекта, който е достъпен като локална променлива с име **session**. Можем да пишем в потока на HTTP отговора чрез методите на **HttpServletResponse** обекта, който е достъпен като локална променлива с име **out**.

Можем да считаме, че JSP страниците са една естествена крачка от развитието на Java-сървлетите като технология, защото разширяват техните възможности и същевременно значително намаляват усилията за създаването на динамични HTML документи.

JSP декларации

JSP декларациите представляват фрагменти програмен код на Java, които се вмъкват директно в кода на генерирания от JSP страницата сървлет. За разлика от скриптлетите, които се вмъкват в тялото на метод, който се вика от метода **service(…)** на класа **HttpServlet**, JSP

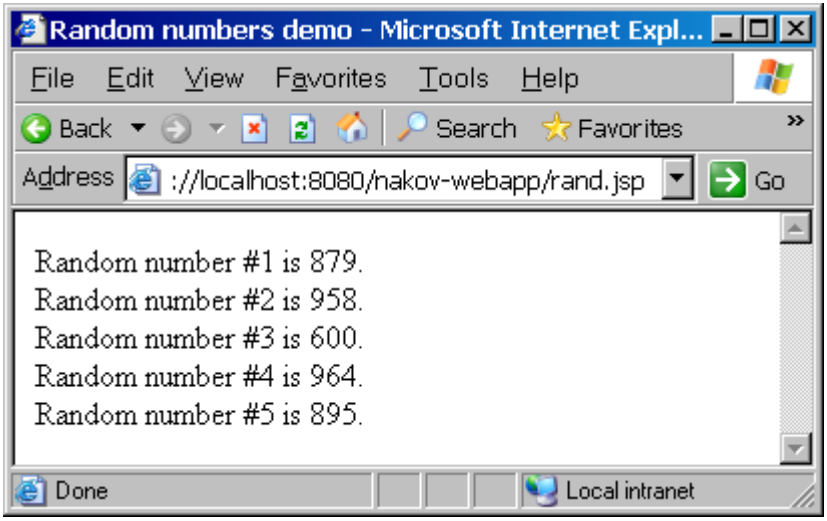
декларациите се вмъкват не в някакъв метод, а директно в тялото на класа. Синтактично JSP декларациите се отделят от статичния HTML текст чрез тага **<%!** … **%>**. Използват се най-често за дефиниране на методи, които след това могат да се извикват от скриптлетите, а също и за деклариране на член-променливи в класа на сървлета, който се получава от даденото JSP. Ето един пример за JSP, което генерира и отпечатва 5 случайни числа, всяко от които е между 0 и 999:

rand.jsp

```
<%!
    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<=5; i++) { %>
            Random number #<%= i %> is
            <%= ""+getRandomNumber(1000) %>.
            <br>
        <% } %>
    </body>
</html>
```

Както се вижда от кода, в този пример с тага **<!%** … **%>** в класа на сървлета се декларира и инициализира член-променлива от тип **java.util.Random** и се добавя метод, 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.

При изпълнение на страницата **rand.jsp** може да се получи примерно такъв резултат:



JSP атрибути

В примера 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 пълните имена на пакетите, към 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 атрибут 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 страницата в сървлет.

Чрез подобен атрибут на JSP документа може да се зададе и content-type-а и encoding на върнатия HTTP отговор. 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" %>
```

Ако искаме да укажем на клиентския Web-браузър, че върнатият от документ трябва да се изобрази на кирилица с българската кодова таблица, трябва да зададем атрибута:

```
<%@ page contentType="text/html;charset=windows-1251" %>
```

Забележете, че след символа "**;**" не трябва да има интервал.

Ако използваме език, който не използва латинската азбука, е полезен и атрибутът **pageEncoding**. С него можем да зададем кодирането, използвано в текущия JSP документ. Например, ако използваме кодиране „UTF-8", трябва да зададем атрибута:

```
<%@ page pageEncoding="UTF-8" %>
```

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" %>
```

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. Целта е в случай на проблем потребителят да не получи 250 реда exception dump, а да му се покаже културно съобщение за грешка и обяснения как да продължи работата си. Всяка страница за обработка на грешки (error page) 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 необработено изключение.

JSP и JavaBeans

JavaBeans представляват reusable софтуерни компоненти, които могат да се манипулират с визуални редактори. На практика те 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;
- имплементират **`Serializable`** интерфейса.

В JSP страниците чрез тага **`<jsp:useBean … />`** могат да се създават и използват JavaBeans. Например тага:

```
<jsp:useBean id="userInfo" class="dataLayer.UserInfo" />
```

декларира инстанция на Java bean от класа **`dataLayer.UserInfo`** с име **`userInfo`**, която е достъпна от всички скриптлети на текущата JSP страница. Тази декларация е почти еквивалентна на обикновеното инстанциране на клас, което в скриптлет може да стане чрез следния код:

```
<% dataLayer.UserInfo userInfo = new dataLayer.UserInfo(); %>
```

За разлика от директното инстанциране на обект от даден клас, тагът **`<jsp:useBean … />`** дава някои 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. Те имат и друго предимство – могат да се използват с JSP изрази. Например чрез следния код можем да зададем стойност на поле в bean:

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

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 кодът

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

зарежда в полето **password** на bean-а `userInfo` стойността, записана в параметъра с име **userPassword** на HTTP 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 механичен труд, необходим за написването на този код.

Използването на Java bean-ове е много полезно за отделяне на логиката от визуализацията в Web-приложенията. Чрез съвместното използване на JavaBeans и JSP страници се дава възможност на Web-дизайнерът да създаде Web-дизайна 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 страницата, като така концентрира вниманието си върху тях, а не върху HTML таговете.

Включване на фрагменти код към JSP страница

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 включим 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 в началото на JSP документа може да се включи следния ред:

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

Той включва съдържанието на файла **menu.jsp** в текущата JSP страница по време на компилацията й. Включеният код може да не е статичен HTML и може да съдържа JSP тагове.

За включване на фрагмент код в текущата JSP страница има и още един подобен таг: **<jsp:include page="relative_url" />**, но той работи малко по-различно. 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.

Има и още една разлика между двата тага. 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.

Пренасочване към друга страница

Още един полезен таг в JSP стандарта е тагът за пренасочване към друга страница

```
<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 **response.sendRedirect(…)** (browser redirection) и **<jsp:forward … />** (server redirection). Методът **response.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 съвсем 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. Сървърът връща като отговор на клиентската заявка не страницата, която Web-браузърът е поискал, а страницата, която се връща при извличане на URL ресурса, към който е извършено пренасочването.

3.10. Сървлет филтри

В тази тема ще разгледаме една много полезна възможност на Java-базираните Web-приложения – да използват филтри при обработката на клиентските заявки.

Какво е сървлет филтър

Сървлет филтрите са Java класове, които служат за прихващане и обработка на заявките, идващи към дадено Web-приложение, както и на отговорите, което то връща в резултат на тези заявки. Те могат да се прикачват към групи сървлети и JSP страници и да променят поведението им.

Когато се извика даден сървлет или JSP, към който има прикачен филтър, заявката се приема първо от филтъра. Той може да я прегледа и да реши дали да позволи извикването на поисканията сървлет или JSP страница или да върне някакъв друг резултат. По този начин на практика може да се „филтрира" обменяната информация между клиентите и Web-приложението.

Някои типични случаи за употреба на филтри са:

- за ограничаване на достъпа до ресурси (чрез парола, по IP адрес или по друг критерий);
- за трансформация на върнатите от сървъра отговори (например за смаляване на всички картинки от дадена директория или за премахване на всички нецензурни думи от даден сайт и подобни);
- за проследяване на заявките към дадено приложение (logging);
- за автоматично прозрачно за програмиста компресиране на информацията.

Във всички тези ситуации сървлет филтрите застават на пътя между клиентския Web-браузър и сървлетите и JSP-тата от Web-приложението и извършват допълнителна обработка на преминаващите през тях заявки и отговори на заявки.

За всеки филтър може да се укаже върху кои ресурси да се прилага, например за всички ресурси на приложението или за всички ресурси от дадена директория или конкретно за даден сървлет или JSP. За един и същ ресурс могат да се прилагат много филтри. В този случай филтрите се изпълняват един след друг и образуват вериги (filter chains).

Сървлет филтрите филтрират информацията, която преминава през тях и на отиване и на връщане. Така една заявка, докато стигне до сървлета или JSP страницата, за която е предназначена, преминава през цялата

верига филтри, които за зададени за нея. След това се изпълнява и върнатия резултат преминава на обратно отново през всички сървлети от веригата и едва след това се връща на клиента. По пътя всеки един от сървлет може да промени заявката или отговора, който преминават през него.

Как се пишат сървлет филтри

За да създадем сървлет филтър е необходимо да напишем клас, който имплементира интерфейса **`javax.servlet.Filter`** и да го опишем в конфигурационния файл на приложението **`web.xml`**, като зададем за кои заявки се отнася той.

Интерфейсът **`javax.servlet.Filter`** има три метода – **`init(…)`**, **`doFilter(…)`** и **`destroy()`**. Методът **`init(…)`** се извиква преди първата заявка към филтъра и му предоставя възможност да извърши първоначални инициализации преди започване на работа. Методът **`destroy()`** се извиква при спиране на Web-приложението и унищожаване на филтъра. В него филтърът трябва да освободи ресурсите, които е използвал.

Най-важният метод, който трябва задължително да се имплементира, е **`doFilter(ServletRequest, ServletResponse, FilterChain)`**. Той приема три параметъра – клиентската заявка, обект за записване на отговора на заявката и обект, представящ веригата от филтри, която следва след този филтър. В този метод трябва да се имплементира логиката на филтъра – специфичните действия, свързани с обработката на клиентската заявка и отговора, получен при изпълнението й.

При обработка на клиентската заявка във метода **`doFilter(…)`** филтърът може да извърши едно от следните три действия:

- да предаде заявката за обработка на следващия филтър във веригата чрез метода **`doFilter(…)`** на **`FilterChain`** параметъра, при което заявката ще се обработи последователно от всички останали филтри и накрая от сървлета или JSP страницата, към която е била оригинално предназначена;
- да пренасочи изпълнението на заявката към някой друг сървлет или JSP страница чрез метода **`sendRedirect(…)`** на **`HttpServletResponse`** класа (към който може да се преобразува **`ServletRequest`** параметъра);
- да запише директно някакъв отговор в **`ServletResponse`** параметъра.

Пример за сървлет филтър

Да си поставим като задача създаването на сървлет филтър, който добавя в края на всяка HTML страница, върната от Web-приложението, някакъв рекламен банер. За целта трябва да имплементираме интерфейса **javax.servlet.Filter**, да прихващаме всички отговори на заявки към Web-приложението и ако те съдържат HTML код, да добавяме в края му рекламния банер. Ето една примерна реализация на такъв филтър:

AdvertismentFilter.java

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

public class AdvertismentFilter implements Filter {
    public void init(FilterConfig aFilterConfig)
            throws ServletException {
    }

    public void doFilter(ServletRequest aRequest,
            ServletResponse aResponse, FilterChain aFilterChain)
            throws IOException, ServletException {
        PrintWriter responseOutput = aResponse.getWriter();
        HttpServletResponse httpResponse =
            (HttpServletResponse) aResponse;
        MemoryResponseWrapper responseWrapper =
            new MemoryResponseWrapper(httpResponse);
        aFilterChain.doFilter(aRequest, responseWrapper);
        String contentType = responseWrapper.getContentType();
        String originalResp = responseWrapper.toString();
        if ((contentType != null) && contentType.toLowerCase().
                startsWith("text/html")) {
            String newResponse = addAdvertisment(originalResp);
            responseOutput.print(newResponse);
        } else {
            responseOutput.println(originalResp);
        }
        responseOutput.close();
    }

    private String addAdvertisment(String aHtmlText) {
        int endOfBodyIndex =
            aHtmlText.toLowerCase().indexOf("</body>");
        String htmlBefore;
        String htmlAfter;
        if (endOfBodyIndex != -1) {
            htmlBefore =
                aHtmlText.substring(0, endOfBodyIndex-1);
            htmlAfter = aHtmlText.substring(endOfBodyIndex);
```

```
        } else {
            htmlBefore = aHtmlText;
            htmlAfter = "";
        }
        String result =
            htmlBefore +
            "\n\n<br><p align='center'>" +
            "<a href='http://www.devbg.org'><img border='0' " +
            "src='http://www.devbg.org/ads/basd-logo.png'>" +
            "</a></p>\n\n" +
            htmlAfter;
        return result;
    }

    public void destroy() {
    }
}

class MemoryResponseWrapper extends HttpServletResponseWrapper {
    private CharArrayWriter mOutput;

    public MemoryResponseWrapper(HttpServletResponse aResponse){
        super(aResponse);
        mOutput = new CharArrayWriter();
    }

    public String toString() {
        String result = mOutput.toString();
        return result;
    }

    public PrintWriter getWriter() {
        PrintWriter printWriter = new PrintWriter(mOutput);
        return printWriter;
    }
}
```

Как работи примерният сървлет филтър

Сървлет филтърът за добавяне на рекламен банер към всяка Web-страница, генерирана от дадено Web-приложение, прихваща отговорите на всички HTTP заявки към приложението и в тези от тях, които съдържат HTML документ, добавя точно преди затварящия таг на тялото му (**</body>**) рекламния банер.

Прихващането на всички HTTP заявки чрез сървлет филтър никак не е трудно. Просто се имплементира интерфейса **javax.servlet.Filter** и в метода му **doFilter(…)** се имплементира обработка на всяка една HTTP заявка.

Прихващането на HTTP отговорите на клиентските заявки, обаче, не е толкова проста работа. За да се прихване клиентският отговор трябва да се изпълни следващият филтър от веригата, като му се подаде обект, в който той да генерира отговора си (на практика този отговор представлява поискания от клиента сървърски ресурс). След това прихванатия отговор на заявката може да се промени и да се запише в изходния поток на HTTP отговора, който филтърът предава на предходния филтър от веригата (или на сървъра, ако няма предходен).

Извикването на следващия филтър от веригата приема като параметри два обекта – **`ServletRequest`** и **`ServletResponse`**. За да се прихване това, което се записва като отговор в **`ServletResponse`** параметъра, трябва да се подаде обект от специален клас, който имплементира интерфейса **`ServletResponse`**, и записва целия отговор в свой вътрешен буфер. Това се прави, защото **`ServletResponse`** интерфейса няма метод за вземане на записания в него отговор. За да се спести писане по имплементацията на всички методи на **`ServletResponse`** интерфейса (защото те никак не са малко), се наследява класа **`HttpServletResponseWrapper`** и в метода му **`getWriter()`** се връща референция към някакъв вътрешен буфер, в случая **`CharArrayWriter`** обект (текстов поток, който съхранява записаната в него текстова информация в масив от символи с динамично-нарастваща дължина). Така сървлетите, които пишат в подадения им **`ServletResponse`**, реално пишат в буфера на класа, който сме им подали.

След изпълнението на всички сървлети по веригата се проверява какъв content-type е върнатият отговор. Ако съдържанието започва с низа "**`text/html`**" (т.е. е HTML документ), в него се намира низа „**`</body>`**" и непосредствено преди него се вмъква HTML код, който визуализира рекламния банер. Ако низът „**`</body>`**" липсва, рекламният банер се слага в самия край на документа. Проверката за типа content-type е важна, защото не е редно да се правят опити да се вмъкват HTML рекламни банери в JPEG изображения, ZIP файлове или други не HTML ресурси.

При някои сървлети и други ресурси примерният сървлет филтър може и да не добавя рекламния банер, но това е защото за тях не е указано, че имат за content-type HTML документ. Ако филтърът добавя банера към всички върнати от Web-приложението ресурси, може да повреди някои от тях, който не е HTML.

Как да инсталираме сървлет филтъра

За да накараме нашият сървлет филтър да работи, трябва да го опишем в конфигурационния файл на Web-приложението **web.xml**. Ето един пример как може да стане това:

web.xml

```
<web-app>
    <filter>
        <filter-name>AdvertismentFilter</filter-name>
        <filter-class>AdvertismentFilter</filter-class>
    </filter>

    <filter-mapping>
      <filter-name>AdvertismentFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Сървлет филтъра се описва чрез име, на което се съпоставя име на клас, а след това по името се съпоставя URL маска, към която се отнася филтъра. Изразът „**/***” означава всички ресурси от приложението. Символът „*****” означава 0 или повече произволни символи.

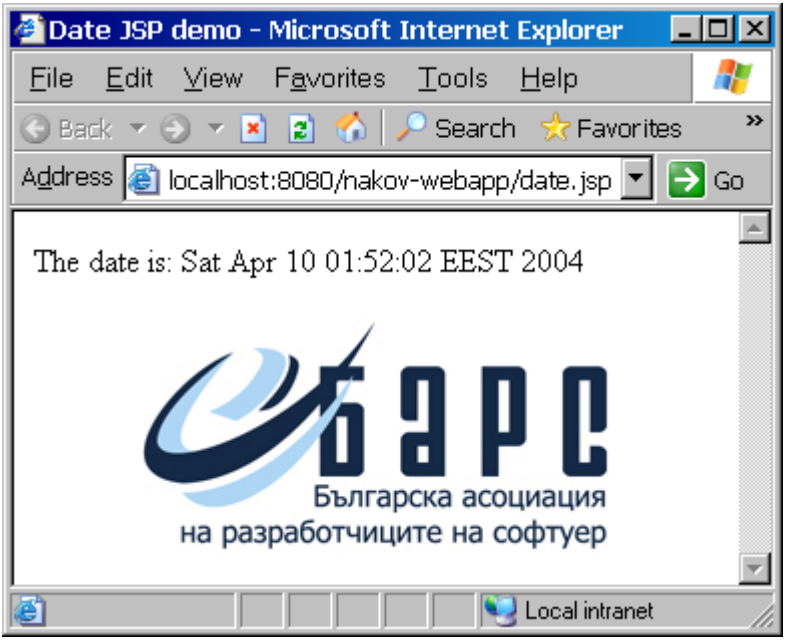
За URL маска може да се сложи и друг израз. Например изразът „**/images/***” означава всички ресурси от директория **images** на Web-приложението. Възможно е и да не се използва маска. Например изразът “**/date.jsp**” означава, че филтърът ще се приложи само и единствено върху ресурса с име „**date.jsp**” от главната директория на Web-приложението.

В нашия случай прилагаме филтъра **AdvertismentFilter** за всички ресурси на Web-приложението, включително и за ресурси, които не съществуват.

Примерният сървлет филтър в действие

За да стартираме сървлет филтъра, трябва да копираме в директория **WEB-INF\classes** на приложението **.class** файловете, които се получават при компилирането му и да го опишем във файла **web.xml**.

Ето как изглежда нашата добра стара JSP страничка за показване на текущата дата (**date.jsp**) след като е преминала през филтъра за добавяне на рекламен банер:


Date JSP demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address localhost:8080/nakov-webapp/date.jsp
Go
The date is: Sat Apr 10 01:52:02 EEST 2004
Българска асоциация
на разработчиците на софтуер
Local intranet

3.11. Тънкости при разработката на Web-приложения с Java

В настоящата тема ще разгледаме структурата на Java-базираните Web-приложения според стандартите на J2EE, а след това ще обърнем внимание на някои тънкости при разработката на Web-приложения без познаването на които няма да можем да разработваме професионални Web-базирани системи.

Структура на J2EE Web-приложенията

J2EE Web-приложенията представляват съвкупност от файлове, които се разполагат в поддиректориите на дадена директория с фиксирана структура. Тази директория се нарича основна (или главна) за Web-приложението и структурата й се задава от J2EE спецификацията. За всеки тип файлове спецификацията определя точно местоположение в рамките на структурата на приложението:

- JSP файловете се разполагат в главната директория на приложението или в поддиректории, създадени от програмиста с цел по-добро структуриране на приложението.
- Настройките на приложението се задават в специалния конфигурационен файл с име **`web.xml`** намиращ се в поддиректория **`WEB-INF`**.
- Класовете, които приложението използва се разполагат в поддиректория **`WEB-INF\classes`**. В същата директория се разполагат и Java сървлетите. Ако класовете имат пакети, за тях се създават съответни поддиректории.
- Java архивите, които могат да съдържат класове, изображения и други ресурси се разполагат в поддиректория **`WEB-INF\lib`**. Най-често в тази **`lib`** директория се разполагат библиотеки с класове, които приложението използва.

Благодарение на тази фиксирана структура на приложението Web-контейнерът знае къде да търси различните файлове, когато му потрябват.

Обикновено всички файлове заедно с цялата структура на директориите се записва в Java архив (ZIP файл) с име името на Web-приложението и с разширение **`.war`**. Например едно приложение за Web-базиран дискусионен форум може да се казва **`forum.war`** и да представлява **`.zip`** файл със следната структура:

```
WEB-INF/
    web.xml
```

```
    classes/
        CommonUtils.class
        DeleteServlet.class
    lib/
        xercesImpl.jar
        xml-apis.jar
header.jsp
login.jsp
logout.jsp
footer.jsp
postMessage.jsp
showMessages.jsp
```

При deploy-ване на приложението в Web-контейнера, който ще го изпълнява (например на сървър Tomcat), файлът, в който то се намира (в нашия случай **forum.war**) се разархивира от сървъра и за приложението се създава виртуална Web директория, достъпна от някакъв URL адрес, например http://www.myserver.com/forum/. Ако не е указано друго, директорията има името на Web-приложението.

Забраняване на кеширането на Web-браузъра

Понеже Web-приложенията работят най-вече с динамично-генерирано съдържание, кешът на браузърите често пъти може да се окаже досаден проблем. Например ако имаме динамична страница за показване на информация, която се променя на всяка секунда, вероятно няма да искаме потребителят да вижда остарели данни заради кеша на браузъра. За да се забрани кеша на браузъра за текущата страница се задават няколко специални полета в хедъра на HTTP отговора. Ето един фрагмент от JSP страница, който указва на браузъра да не кешира документа, който получи:

```
<%
    response.setHeader("Pragma", "No-cache");
    response.setDateHeader("Expires", 0);
    response.setHeader("Cache-Control", "no-cache");
%>
```

Препоръчва се трите посочени реда да се използват заедно заради съвместимост с всички браузъри.

Проблеми със специалните символи в HTML

Да разгледаме следния фрагмент от JSP страница:

```
<% String name = request.getParameter("name"); %>
Welcome, <%= name %>!
```

В кода има един сериозен проблем. Ако параметърът **name** има стойност **<font color="red">**, вместо да се отпечата поздрав с името на потребителя, ще се отпечата HTML таг, който задава червен цвят за остатъка от HTML документа. Ефектът може да бъде дори много по-страшен ако потребителят въведе за име на потребител следното:

```
<script language="JavaScript">while (1) alert("Bug!");</script>
```

Ако не се досещате какво ще се случи, пробвайте. При повечето Web-браузъри ефектът ще е неприятен: постоянно ще излиза съобщение „Bug!" и браузърът дори няма да може да бъде затворен.

При една сложна Web-базирана система е възможно потребителят да въвежда нещо и то да отива директно при някои оператор, който го обработва. Тогава неприятният ефект няма да се стовари върху потребителя, който го е предизвикал, а върху оператора. Ако нападателят е достатъчно хитър и достатъчно злонамерен, могат да се случат дори още по-лоши неща. Например на машината на оператора може да се появи съобщение, че сесията му е изтекла и HTML форма, в която да си въведе паролата, за да му бъде възобновена сесията. След това, естествено, въведената парола може свободно да бъде изпратена при нападателя. Такъв род проблеми със сигурността са известни като „cross-site scripting " уязвимости и потенциално съществуват при всички езици и технологии за динамично генериране на HTML.

Справяне с проблема със специалните символи в HTML

Очевидно проблемът е доста сериозен и застрашава нормалната работа на системата. Да помислим как можем да го решим. Единият начин да се справим е като филтрираме някои непозволени символи, винаги, когато приемаме данни, идващи от потребителя. Това не винаги е възможно, потребителят може да иска да изпрати някакъв HTML документ като нормална част от работата си. Трябва ни друго решение.

Правилният начин за справяне с проблема със специалните символи в HTML, е чрез заместването им с еквивалентни последователности от символи, които не съдържат специални за HTML символи. Такова преобразование се нарича **escaping** (ескейпване). Има различни видове ескейпване.

Ако искаме да ескейпнем текст, който да поставим като параметър в даден URL адрес, трябва да използваме ескейпването „URL encode", което замества символа *интервал* със символа „+" или с последователността от символи „**%20**", символа *въпросителен знак* – с

последователността „**%3F**" и т.н. За такова ескейпване в Java може да се използва класа метода **encode** на класа **java.net.URLEncoder**.

Ако искаме да поставим безопасно текст в HTML тага **<textarea>** трябва да избегнем единствено символите „**<**" и „**&**" като ги заменим с последователностите „**<**" и „**&**".

Ако искаме да поставим безопасно стойност на текстово поле или стойност на атрибут на HTML таг, трябва да избегнем символите *кавичка*, *апостроф* и „**&**" като ги заместим съответно с „**"**", „**'**" и „**&**".

Ако искаме да поставим безопасно текст в тялото на HTML документ, трябва да осигурим избягването на символите „**<**", „**>**", *нов ред*, *интервал*, *табулация* и „**&**" съответно с последователностите „**<**", „**>**", „**
", „ **", „**   **" и „**&**". Считаме, че табулацията се разглежда като 4 последователни интервала.

За последните три случая в Java няма стандартен клас или метод, който да извършва ескейпването. Колкото и странно да изглежда, колкото и този проблем да присъства във всички Java-базирани Web-приложения, нито в Servlet API спецификацията, нито в друг стандартен за J2EE или J2SE клас няма метод за ескейпване на HTML текст.

Единствената възможност, която ни остава, е да използваме собствен метод за HTML ескейпване. Трябва много да внимаваме да не пропуснем някой символ или да не ескейпнем някой символ грешно, защото това ще наруши сигурността на всички приложения, които използват нашия код. Ето една коректна реализация:

```
/**
 * Escapes given text for placing it in the HTML body. If
 * you need escaping for placing text in an attribut value,
 * you should remove the escaping for the "\n" character.
 *
 * (c) Svetlin Nakov, 2004 - http://www.nakov.com
 */
public static String htmlEscape(String aText) {
    if (aText == null) {
        return "";
    }
    StringBuffer escapedText = new StringBuffer();
    for (int i=0; i<aText.length(); i++) {
        char ch = aText.charAt(i);
        if (ch == '\'')
            escapedText.append("&#39;");
        else if (ch == '\"')
            escapedText.append("&#34;");
        else if (ch == '<')
```

```
            escapedText.append("&lt;");
        else if (ch == '>')
            escapedText.append("&gt;");
        else if (ch == '&')
            escapedText.append("&amp;");
        else if (ch == '\n')
            escapedText.append("<br>\n");
        else if (ch == ' ')
            escapedText.append(" ");
        else if (ch == '\t')
            escapedText.append("    ");
        else
            escapedText.append(ch);
    }
    String result = escapedText.toString();
    return result;
}
```

Този сорс код може да се използва за ескейпване на текст, който ще се поставя директно в тялото на HTML документ. При ескейпване на текст за поставяне като стойност на атрибут от HTML таг или в тялото на **<textarea>**, трябва да се премахне ескейпването на символа *нов ред*, защото ще причини проблеми. Всичко останало може да се запази.

Липсата на ескейпване или неправилното ескейпване може да причини проблеми със сигурността и стабилността на нашето Web-приложение. За да се предпазим от такива проблеми, винаги, когато генерираме динамичен HTML, трябва да спазваме следните правила:

- Винаги трябва да ескейпваме преди директно отпечатване на текст в динамични HTML документи!
- Винаги трябва да използваме правилния тип ескейпване – URL encode, HTML escaping или HTML escaping за стойност на атрибут или **<textarea>**.
- Текстът трябва да се ескейпва непосредствено преди отпечатването му в документа! Не трябва да ескейпваме текста още при получаването му, защото рискуваме да се получи проблема „двойно ескейпване”.

Използвайки метода **htmlEscape(…)**, можем да поправим проблемния JSP фрагмент от примера по-горе по следния начин:

```
<% String name = request.getParameter("name"); %>
Welcome, <%= htmlEscape(name) %>!
```

Сега вече каквото и да изпрати потребителя като стойност на параметъра **name**, няма да наруши правилната работа на JSP страницата.

Кирилицата в сървлети и JSP страници

Основният проблем с кирилицата в сървлетите и JSP страниците идва от това, че при протокола HTTP заявката и отговорът обикновено са текстове, при които един символ се представя с един байт. Понеже в Java един символ се представя с Unicode (2 байта), възниква проблемът как да се преобразува малкото множество на еднобайтовите символи в голямото множество на Unicode символите. Очевидно съответствието не е еднозначно. За задаване на такова съответствие се използват т. нар. схеми за кодиране на символите (character encodings).

Повечето схеми за кодиране задават съответствия между част от еднобайтовите символи и част от Unicode символите. При преобразуването от Unicode към някоя кодираща схема или обратното всички символи, за които в схемата няма дефинирано съответствие, се заменят със символа „?” (байт със стойност 63).

Често пъти при използване на кирилица в сървлети и JSP страници вместо кирилица излизат въпросителни знаци. Това се дължи на неправилната кодираща схема, която се използва. За представяне на кирилица най-често се използва стандартната кодираща схема „windows-1251”. Ако не бъде указано да бъде използвано точно тя, често пъти възникват проблеми.

Проблемите с кирилицата при работа със сървлети и JSP страници са два – проблем с кодирането на HTTP заявката и проблем с кодирането на отговора на HTTP заявката.

Задаване на кодирането на HTTP заявката

В сървлети и JSP страници, които приемат параметри от HTTP заявката (например чрез метода **`getParameter(…)`** на **`HttpServletRequest`** класа) може да възникне проблем с кирилицата и в резултат на това в стойностите на параметрите да има въпросителни знаци на мястото на всички букви от кирилицата. Този проблем се решава чрез задаване на кодирането на HTTP заявката посредством израза:

```
<%
    request.setCharacterEncoding("cp1251");
%>
```

Задаването на схемата за декодиране на HTTP заявката трябва да е първото нещо, което прави един сървлет или JSP страница. След първото извикване на метода **`getParameter()`** на **`request`** обекта задаването на схема за декодиране няма никакъв ефект.

Въпреки всичко в някои ситуации може горният израз да не реши проблема с кирилицата. Методът **setCharacterEncoding(…)** по принцип се отнася до тялото на HTTP заявката и затова действа за всички параметри, изпратени по HTTP POST метод, но при заявки по метод HTTP GET е възможно да няма ефект. В такъв случай трябва да или да се премине към използване на POST заявки или да се променят стандартните настройките на Web-контейнера, за да се установи кодиране по подразбиране windows-1251.

Задаване на кодирането на HTTP отговори на заявки

Проблемът с неправилната схема на кодиране на HTTP отговора може да бъде решен по подобен начин – като бъде зададена подходяща схема за кодиране. В JSP страница за да укажем, че искаме отговорът на HTTP заявката да бъде разглеждан като текст на кирилица с кодиране windows-1251, трябва да зададем следния JSP атрибут:

```
<%@ page contentType="text/html;charset=windows-1251" %>
```

Ако искаме да укажем същото от сървлет, можем да го направим така:

```
response.setContentType("text/html;charset=windows-1251");
```

Задаването на content-type трябва да е първото нещо, което сървлетът прави, защото след като започне писането в потока, свързан с отговора на заявката, това вече е невъзможно.

Не винаги е възможно да зададем content-type на една JSP страница, защото ако една страница включва друга, се допуска само едната от двете да използва директивата **<%@page contentType="…" %>**. В такива случаи за страницата, която се включва в другата трябва да използваме директивата:

```
<%@ page pageEncoding="windows-1251" %>
```

С нея указваме на сървлет-контейнера какво е кодирането на страницата без да задаваме content-type.

3.12. Цялостен пример за Web-приложение

След като вече сме добре запознати с технологиите на Java сървлетите, Java Server Pages (JSP) и концепциите за създаване на Java-базирани Web-приложения, ще си поставим за задача да създадем едно цялостно Web-приложение, с което да демонстрираме как можем да комбинираме досегашните си знания за целите на един малък проект.

Web-приложението „Мини форум”

Да си поставим за задача разработката на много прост дискусионен форум. От гледна точка на потребителя приложението трябва да има две страници – едната за влизане във форума, а другата за четене на съобщенията, добавяне на нови съобщения и изтриване на ненужни съобщения.

Форумът трябва да позволява много потребители да работят едновременно. За влизане във форума трябва да се изисква потребителско име и парола. Ако въведеното име съвпада с въведената парола, системата трябва да пуска потребителя на страницата със съобщенията. За простота няма да има администратор, който да конфигурира валидните потребители и пароли.

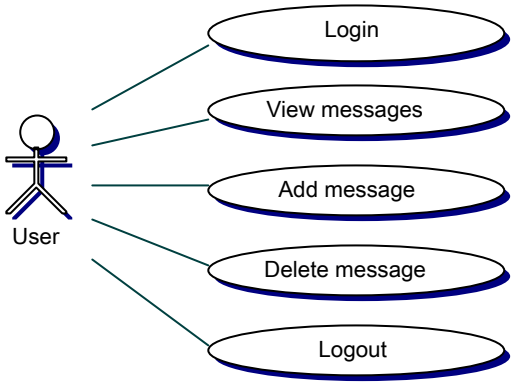
В страницата за съобщенията трябва да се извеждат в таблица всички съобщения, като след всяко трябва да има hyperlink за изтриване и да има форма за добавяне на ново съобщение. Съобщенията трябва да се състоят от тема и съдържание. За леснота съобщенията могат да се пазят само в паметта на приложението, т.е. се губят при рестартиране на сървъра. Системата не трябва да позволява достъп до форума на неоторизирани потребители, който не са влезли през началната страница.

Системата трябва да показва в долната част на всяка страница от приложението името на потребителя, който е влязъл във форума и да му позволява да излезе по свое желание (logout).

Анализ на изискванията

Поставената задача не е много сложна и е далеч от реален проект, но ние ще я разгледаме като един добър пример за Web-приложение. За съжаление познанията ни от настоящата книга не са достатъчни за да направим приложението наистина реално, но и целта на книгата не е да научим всичко за Java (не сме се запознали с релационните бази от данни и достъпа до тях от Java и не сме разгледали другите средства на J2EE платформата, като например EJB, JMS и още много други неща).

Все пак нека да анализираме задачата и да дадем едно примерно решение. Имаме следните 5 случаи на употреба (use cases):



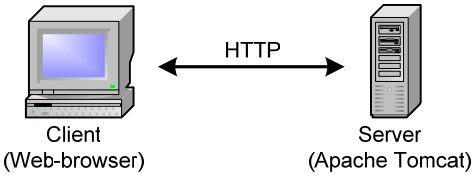
Потребителите могат да преглеждат, добавят и изтриват съобщения, само ако са успешно автентикирани. При опит за достъп до някоя страница първо трябва да се проверява дали текущият потребител е автентикиран и ако не е, да се препраща към страницата за влизане в системата. След успешна автентикация потребителят трябва да се препраща обратно към поисканата от него преди това страница.

Преглеждането, добавянето и изтриването на съобщения може да става от една и съща страница – главната страница на форума.

Приложението трябва да се съобразява с възможността няколко потребители едновременно да разглеждат форума, да добавят съобщения и да изтриват съобщения и не трябва да създава проблеми с конкурентния достъп (например ако двама потребители в един и същ момент се опитат да изтрият едно и също съобщение).

Архитектурен план на Web-приложението

За нашето приложение ще използваме класическия двуслоен модел „клиент-сървър”:

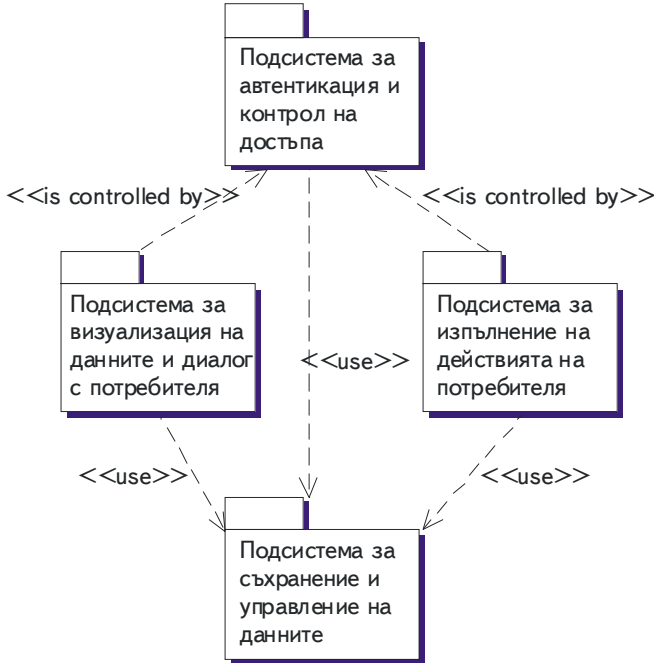


Клиентът ще е потребителският Web-браузър, а сървърът ще е нашето Web-приложение, работещо върху сървлет-контейнера.

За да улесним разработката и поддръжката на приложението, ще го разделим логически на няколко подсистеми:

- подсистема за съхранение и управление на данните;
- подсистема за автентикация и контрол на достъпа;
- подсистема за визуализация на данните и диалог с потребителя;
- подсистема за изпълнение на действията на потребителя.

На диаграмата е показано как тези подсистеми си взаимодействат и зависят една от друга:



Дизайн на Web-приложението

Ще използваме класическия програмен модел **MVC** (Model-View-Controller). При класическият MVC модел класовете, които изграждат отделните подсистеми, се разделят на три типа – **model** (класове, които представят данните и осигуряват достъп до тях), **view** (класове, които визуализират данните и предоставят потребителския интерфейс) и

controller (класове, които управляват действията на потребителите и бизнес процесите в приложението).

Подсистема за съхранение и управление на данните

Подсистемата за съхранение и управление на данните се отнася към **model** частта от MVC модела. Тя има за цел да съхранява данните на приложението и да осигурява достъп до тях. Тези данни се състоят от данните за потребителите и данните за съобщенията във форума.

Подсистемата позволява задаване на текущ потребител, извличане на текущия потребител, взимане на всички съобщения, добавяне на съобщение и изтриване на съобщение.

Тя съхранява данните за текущия потребител в неговата сесията, а данните за съобщенията от форума – в контекста на приложението (за да са общи за всички потребители).

Понеже съобщенията от форума са общи за всички потребители, подсистемата трябва да се грижи за правилното управление на конкурентния достъп до тях.

Подсистема за автентикация и контрол на достъпа

Подсистемата за автентикация и контрол на достъпа се отнася към **controller** частта от MVC модела и осигурява сигурността на приложението. Тя управлява процесите на автентикация на потребители, оторизация на достъпа до ресурсите на приложението, влизането в системата (login) и излизането от системата (logout).

Подсистемата филтрира всички заявки към Web-приложението и пропуска само тези от тях, които идват от автентикирани потребители или са свързани с влизането на нов потребител.

Данните за автентикираните потребители се съхраняват в тяхната сесия посредством подсистемата за съхранение и управление на данните.

Подсистема за визуализация на данните и диалог с потребителя

Подсистемата за визуализация на данните и диалог с потребителя се отнася към **view** частта на MVC архитектурата. Тя осигурява потребителския интерфейс на Web-приложението. Тя се състои от множество JSP страници, които предоставят потребителски интерфейс за влизане в системата, за разглеждане на съобщенията от форума, за добавяне на нови съобщения, за изтриване на съобщения и за излизане от приложението.

Подсистемата си взаимодейства пряко с подсистемата за изпълнение на действията на потребителя, а достъпът до отделните елементи от потребителския интерфейс се контролира от подсистемата за автентикация и контрол на достъпа.

Подсистема за изпълнение на действията на потребителя

Подсистемата за изпълнение на действията на потребителя се отнася към **controller** частта на MVC архитектурата. Тя обработва и изпълнява заявените от потребителите действия, идващи от подсистемата за потребителски интерфейс. Тези действия включват добавяне на ново съобщение във форума и изтриване на съществуващо съобщение.

Имплементация на Web-приложението

За да имплементираме Web-приложението е необходимо да имплементираме четирите подсистеми, които идентифицирахме в процеса на проектирането му.

Всички класове от приложението ще разполагаме в пакета **miniforum**. За подсистемите ще използваме подпакети на този пакет, за да ги отделим логически една от друга.

Интерфейсът **miniforum.IConstants** съдържа най-разнообразни константи и се използва от всички подсистеми на приложението. Ето неговият сорс-код:

IConstants.java

```
package miniforum;

public class IConstants {
    public static final String LOGIN_URL = "doLogin";
    public static final String LOGOUT_URL = "doLogout";
    public static final String EMPTY_URL = "";
    public static final String LOGIN_FORM = "login.jsp";
    public static final String MAIN_FORM = "main.jsp";

    public static final String CURRENT_USER = "CURRENT_USER";
    public static final String LAST_ERROR = "LAST_ERROR";
    public static final String ORIGINAL_URL = "ORIGINAL_URL";
    public static final String FORUM_MESSAGES = "MESSAGES";

    public static final String USER_PARAM = "username";
    public static final String PASSWORD_PARAM = "password";
    public static final String ID_PARAM = "id";
    public static final String SUBJECT_PARAM = "subject";
    public static final String CONTENTS_PARAM = "contents";
}
```

За какво се използва всяка една от дефинираните константи ще разберем по-нататък докато разглеждаме имплементацията на подсистемите на приложението.

Подсистема за съхранение и управление на данните – имплементация

Ще имплементираме подсистемата за съхранение и управление на данните като съвкупност от Java класове, разположени в пакета **`miniforum.data`**. Да разгледаме сорс-кода на тези класове.

Започваме с класа **`miniforum.data.UserUtils`**:

UserUtils.java

```
package miniforum.data;

import miniforum.IConstants;

import javax.servlet.http.HttpSession;

public class UserUtils {
    public static String getCurrentUser(HttpSession aSession) {
        String currentUser = (String)
            aSession.getAttribute(IConstants.CURRENT_USER);
        return currentUser;
    }

    public static void setCurrentUser(HttpSession aSession,
            String aUserName) {
        aSession.setAttribute(
            IConstants.CURRENT_USER, aUserName);
    }
}
```

Този клас предоставя възможност за извличане на активния потребител от текущата сесия и за записване на текущия потребител в нея. Той няма нужда от синхронизация, защото не осъществява конкурентен достъп до общи ресурси, понеже сървлет-контейнерът се грижи **`HttpSession`** обекта да е уникален за всеки потребител.

Следващият клас е класът **`Message`**:

Message.java

```
package miniforum.data;

public class Message {
    private long mID;
    private String mUser;
    private String mSubject;
```

```
    private String mContents;

    private static long mSequenceNumber = 0;

    public Message() {
        // Assign an unique ID to the newly create message
        synchronized (Message.class) {
            mSequenceNumber++;
            mID = mSequenceNumber;
        }
    }

    public long getID() {
        return mID;
    }

    public String getUser() {
        return mUser;
    }

    public void setUser(String aUser) {
        mUser = aUser;
    }

    public String getSubject() {
        return mSubject;
    }

    public void setSubject(String subject) {
        mSubject = subject;
    }

    public String getContents() {
        return mContents;
    }

    public void setContents(String contents) {
        mContents = contents;
    }
}
```

Класът **miniforum.data.Message** енкапсулира в себе си данните за едно съобщение от форума – автор, тема и съдържание на съобщение.

Понеже съобщенията трябва да се идентифицират уникално по някакъв начин, класът има грижата при създаването им да им съпоставя по едно уникално 64-битово число (**mID**). Без такава уникална идентификация, ще има трудности при реализацията на изтриване на съобщения. За съпоставянето на уникален идентификатор при създаването на всяко съобщение класът използва статичен брояч, който се увеличава с

единица при всяко следващо съобщение. Понеже е възможно няколко съобщения да бъдат създавани в един и същ момент от различни нишки, при достъпа до брояча се извършва синхронизация по статичния монитор на класа **Message**.

Достъпът до член-променливите на класа не е синхронизиран, защото се счита, че след като е създадено едно съобщение, то не се променя и следователно няма да настъпи момент, в който няколко нишки се опитват едновременно да променят едно и също съобщение. За нашия случай това наистина е така и затова сме си спестили синхронизацията.

Следва класът **miniforum.data.MessageUtils**, който управлява достъпа до съобщенията във форума:

MessageUtils.java

```
package miniforum.data;

import miniforum.data.Message;
import miniforum.IConstants;

import javax.servlet.ServletContext;
import java.util.ArrayList;

public class MessageUtils {
    public static synchronized Message[] getForumMessages(
            ServletContext aApplication) {
        ArrayList msgs = (ArrayList) aApplication.getAttribute(
            IConstants.FORUM_MESSAGES);
        if (msgs == null) {
            msgs = new ArrayList();
            aApplication.setAttribute(
                IConstants.FORUM_MESSAGES, msgs);
        }

        Message[] copyOfMsgs =
            (Message[]) msgs.toArray(new Message[]{});
        return copyOfMsgs;
    }

    public static synchronized void addForumMessage(
            ServletContext aApplication, Message aMessage) {
        if (aMessage.getSubject() == null ||
                aMessage.getSubject().length() == 0 ||
                aMessage.getContents() == null ||
                aMessage.getContents().length() == 0) {
            throw new IllegalArgumentException("Invalid msg!");
        }
        ArrayList msgs = (ArrayList) aApplication.getAttribute(
            IConstants.FORUM_MESSAGES);
        msgs.add(aMessage);
```

```
    }

    public static synchronized void deleteForumMessage(
            long aID, ServletContext aApplication) {
        ArrayList msgs = (ArrayList) aApplication.getAttribute(
            IConstants.FORUM_MESSAGES);
        for (int i=0; i<msgs.size(); i++) {
            Message msg = (Message) msgs.get(i);
            if (msg.getID() == aID) {
                // Message found. Delete it
                msgs.remove(i);
                return;
            }
        }
        throw new IllegalArgumentException("Invalid msg ID!");
    }
}
```

Класът позволява получаване на всички съобщения, добавяне на съобщение и изтриване на съобщение.

Всички съобщения се съхраняват в контекста на Web-приложението и затова методите за достъп до тях изискват да им се подава този контекст (**`ServletContext`**) като параметър. Реално съобщенията стоят в масив с променлива дължина (**`java.util.ArrayList`**), който се съхранява като атрибут под ключ **`IConstants.FORUM_MESSAGES`**.

Поради възможността много потребители едновременно да се опитват да четат или променят съобщенията във форума, се налага да се погрижим за синхронизацията на достъпа до тях.

Всички методи за достъп са синхронизирани по статичния монитор на класа **`MessageUtils`**. Това, обаче съвсем не е достатъчно за безопасната работа на много потребители едновременно.

Ако методът, който връща всички съобщения просто извлича от сесията **`ArrayList`** обекта и директно го връща на извикващия, може да възникне следния проблем: Взимаме всички съобщения и започваме в цикъл да ги отпечатваме. По същото време, докато се изпълнява този цикъл, някой друг потребител изтрива първото съобщение от същия този **`ArrayList`** обект. В резултат от това индексите на всички съобщения намаляват с единица и цикълът за отпечатване на съобщенията прескача някое от съобщенията без то да е било изтрито. Първото съобщение си остава отпечатано, а някое от следващите се изгубва. При подходящо стечение на обстоятелствата може да се получи и изключение **`ArrayIndexOutOfBounds`** при опит за достъп до последния елемент на масива, защото броят на елементите са намалели преждевременно с един.

Описаният сценарий е един от класическите проблеми при конкурентен достъп до общи данни. Той може да бъде решен по два начина – или като се синхронизира цикъла по съобщенията с достъпа до тях, за да не се допуска някой да ги промени по време на изпълнението на този цикъл, или при получаване на съобщенията да не се връща **`ArrayList`** обекта, който реално ги съхранява, а копие от него. За простота използваме втория подход – при извличане на всички съобщения правим в един масив копие на референциите към тях и връщаме него. Това решава проблема.

Подсистема за автентикация и контрол на достъпа – имплементация

Ще имплементираме подсистемата за автентикация и контрол на достъпа като съвкупност от Java класове, разположени в пакета **`miniforum.action`**. За ограничаване на достъпа до приложението на потребители, които не са преминали успешна автентикация, ще използваме специален сървлет филтър. За управлението на login и logout заявките ще използваме сървлети. Да разгледаме сорс-кода на подсистемата:

AuthenticationFilter.java

```
package miniforum.action;

import miniforum.data.UserUtils;
import miniforum.IConstants;

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

public class AuthenticationFilter implements Filter {

    public void init(FilterConfig aFilterConfig)
            throws ServletException {
    }

    public void doFilter(ServletRequest aRequest,
            ServletResponse aResponse, FilterChain aFilterChain)
            throws IOException, ServletException {
        HttpServletRequest httpRequest =
            (HttpServletRequest) aRequest;
        HttpServletResponse httpResponse =
            (HttpServletResponse) aResponse;
        String requestedPage = getRequestedPage(httpRequest);

        if (requestedPage.equals(IConstants.EMPTY_URL)) {
```

```
            // Accessing root directory redirects to login form
            httpResponse.sendRedirect(IConstants.LOGIN_FORM);
            return;
        }

        if (IConstants.LOGIN_FORM.equals(requestedPage) ||
                IConstants.LOGIN_URL.equals(requestedPage) ||
                IConstants.LOGOUT_URL.equals(requestedPage)) {
            // Accessing login/logout is always permitted
            aFilterChain.doFilter(aRequest, aResponse);
            return;
        }

        HttpSession session = httpRequest.getSession();
        boolean authenticated =
            (UserUtils.getCurrentUser(session) != null);
        if (authenticated) {
            // Authenticated user. Permit the request
            aFilterChain.doFilter(aRequest, aResponse);
        } else {
            // Not authenticated user. Redirect to login form
            session.setAttribute(IConstants.LAST_ERROR,
                "Поисканата страница изисква автентикация. " +
                "Моля първо влезте в системата!");
            session.setAttribute(IConstants.ORIGINAL_URL,
                requestedPage);
            httpResponse.sendRedirect(IConstants.LOGIN_FORM);
        }
    }

    private String getRequestedPage(
            HttpServletRequest aHttpRequest) {
        String url = aHttpRequest.getRequestURI();
        int firstSlash = url.indexOf("/",1);
        String requestedPage = null;
        if (firstSlash != -1) requestedPage =
            url.substring(firstSlash + 1, url.length());
        return requestedPage;
    }

    public void destroy() {
    }
}
```

Класът **miniforum.action.AuthenticationFilter** е много важен за сигурността на приложението. Той представлява сървлет филтър, който контролира достъпа до всички ресурси в приложението. През него преминават всички HTTP заявки и той преценява дали да ги пропусне или не.

Филтърът работи по следния начин: При извикване първо извлича името на искания ресурс, а след това анализира този ресурс и потребителя, който го е поискал, за да прецени дали да му позволи достъп.

Ако заявката е за достъп към основната директория на Web-приложението без да е указано име на ресурс се извършва пренасочване към HTML формата за влизане във форума (**`login.jsp`**).

При опит за достъп до формата за влизане във форума (**`login.jsp`**) или до сървлета, който обработва заявките за влизане (**`doLogin`**) или до сървлета за изход от форума (**`doLogout`**) филтърът позволява достъпа (чрез извикване следващия филтър във веригата). Това е съвсем правилно поведение, защото тези ресурси от приложението трябва да са публично достъпни.

След това филтърът взима потребителската сесия и проверява дали потребителят, който е направил заявката е влязъл в системата (дали е автентикиран). Ако е автентикиран, му се позволява достъп до искания ресурс.

При опит за достъп от неавтентикиран потребител се прави следното: в сесията на текущия потребител (в специален атрибут с име **`IConstants.LAST_ERROR`**) се задава съобщение за грешка, което ще бъде показано на първата страница от приложението, на която потребителя попадне. След това се запомня ресурсът, който потребителят е поискал (в атрибут с име **`IConstants.ORIGINAL_URL`**) и потребителят се прехвърля към формата за влизане в системата. Поисканият ресурс преди пренасочването към страницата за влизане във форума се запомня с цел след влизането потребителят да бъде пренасочен към нея.

Чрез проверката на всеки поискан ресурс от приложението филтърът пази от непозволен достъп цялото приложение и позволява нормален достъп само на автентикираните потребители. Този начин на имплементиране на контрола върху достъпа е за предпочитане пред проверката за автентикация във всяка JSP страница или сървлет, защото при другия начин е възможно някой ресурс случайно да бъде забравен да бъде защитен и да възникне опасност за сигурността. Другото предимство на сървлет филтрите е че защитават всички ресурси на приложението, включително статични страници, картинки и други файлове, които не са нито сървлети, нито JSP-та.

Да разгледаме как се извършва автентикацията и влизането на нов потребител в системата:

LoginServlet.java

```
package miniforum.action;

import miniforum.data.UserUtils;
import miniforum.IConstants;

import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.IOException;

public class LoginServlet extends HttpServlet {
    protected void doPost(HttpServletRequest aRequest,
            HttpServletResponse aResponse)
            throws ServletException, IOException {
        aRequest.setCharacterEncoding("cp1251");
        String userName =
            aRequest.getParameter(IConstants.USER_PARAM);
        String password =
            aRequest.getParameter(IConstants.PASSWORD_PARAM);
        HttpSession session = aRequest.getSession();
        if (isUserValid(userName, password)) {
            // User valid. Redirect to the last requested page
            UserUtils.setCurrentUser(session, userName);
            String targetURL = (String)
                session.getAttribute(IConstants.ORIGINAL_URL);
            session.removeAttribute(IConstants.ORIGINAL_URL);
            if (targetURL == null) {
                targetURL = IConstants.MAIN_FORM;
            }
            aResponse.sendRedirect(targetURL);
        } else if (userName != null) {
            // User invalid. Redirect to login form
            session.setAttribute(IConstants.LAST_ERROR,
                "Невалиден потребител или парола!");
            aResponse.sendRedirect(IConstants.LOGIN_FORM);
        }
    }

    public static boolean isUserValid(String aUserName,
            String aPassword) {
        boolean valid =
            (aUserName != null) && (aPassword != null) &&
            (aUserName.length() > 0) &&
            (aUserName.equals(aPassword));
        return valid;
    }
}
```

Сървлетът за влизане в системата не е сложен. Той очаква да бъде извикан от HTTP POST заявка и при стартиране взима подадените му

като параметри потребителско име и парола. След това проверява дали потребителят е валиден и дали паролата съответства на този потребител.

За простота в примерното приложение за валидни се считат всички потребители, чиято парола съвпада с потребителското им име. В реална ситуация тази проверка може да се направи по друг начин, така че да извършва проверка за валидността на потребителя в база данни, в някакъв файл или по някакъв друг начин.

При успешна автентикация в сесията се регистрира потребителското име, с което е протекла автентикацията и браузърът на потребителя се препраща към URL адреса, който потребителят е поискал преди да бъде пренасочен към страницата за влизане в системата (той се взема от атрибута **`IConstants.ORIGINAL_URL`** от сесията). Ако такъв адрес не е наличен, това означава, че потребителят е дошъл директно на страницата за влизане в системата без да се е опитвал преди това да зареди друга страница от приложението. В този случай се извършва пренасочване към основната страница на приложението (**`main.jsp`**).

Ако автентикацията е неуспешна, в атрибута **`IConstants.LAST_ERROR`** се поставя съобщение за грешка, което да бъде показано на потребителя при следващата заредена страница от него страница от приложението и след това браузърът му се пренасочва към формата за влизане в системата (**`login.jsp`**).

За да се допускат и потребители с имена на кирилица, в началото на сървлета се задава кодираща схема за HTTP заявката "**`cp1251`**".

Да видим как се извършва излизането от системата:

`LogoutServlet.java`

```
package miniforum.action;

import miniforum.IConstants;

import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.IOException;

public class LogoutServlet extends HttpServlet {
    protected void doGet(HttpServletRequest aRequest,
            HttpServletResponse aResponse)
            throws ServletException, IOException {
        HttpSession session = aRequest.getSession();
        session.invalidate();
        aResponse.sendRedirect(IConstants.LOGIN_FORM);
    }
}
```

Сървлетът за излизане от системата е изключително прост. Той обявява текущата потребителска сесия за невалидна, с което изтрива всичко, записано в нея и след това пренасочва браузъра на потребителя към страницата за влизане в системата (**login.jsp**).

Подсистема за визуализация на данните и диалог с потребителя – имплементация

Ще имплементираме подсистемата за визуализация на данните и диалог с потребителя като съвкупност от JSP страници, разположени в главната директория на Web-приложението.

Подсистемата трябва да предоставя на потребителя форма за влизане в системата, форма за показване на съобщенията от форума, форма за добавяне на съобщение и форма за изтриване на съобщение.

Понеже искаме всяка страничка от Web-приложението да има заглавна част, в която да пише името на приложението и завършваща част, в която да пише името на потребителя, който е влязъл в момента, е най-добре да отделим заглавната част и завършващата част в отделни файлове, които да се включват към всяка страница. Ето как изглежда кодът на заглавната част:

header.jsp

```
<%@ page import="miniforum.HtmlUtils,
                 miniforum.IConstants"%>
<%@ page pageEncoding="windows-1251" %>

<%-- Disable browser caching --%>
<%
    response.setHeader("Pragma", "No-cache");
    response.setDateHeader("Expires", 0);
    response.setHeader("Cache-Control", "no-cache");
%>

<table border="0" width="100%">
    <tr><td align="center" bgcolor="#66CCFF">
        Мини форум - (c) Светлин Наков, 2004
    </td></tr>
<%
    String errorMsg = (String)
        session.getAttribute(IConstants.LAST_ERROR);
    if (errorMsg != null) {
%>
    <tr><td align="center">
        <br>
        <font color="red"><b>
            Грешка: <%=HtmlUtils.htmlEscape(errorMsg)%>
```

```
            </b></font>
            <br>
        </td></tr>
<%
    session.removeAttribute(IConstants.LAST_ERROR);
}
%>
</table>
```

Заглавната част **header.jsp**, която се включва в началото на всяка JSP страница, започва декларация за включване на класовете **HtmlUtils** и **IConstants**, които се използват по-нататък в нея.

Класът **HtmlUtils** се използва, защото в него е дефиниран метода **htmlEscape(…)**, който замества всички специални за езика HTML символи с еквивалентна последователност от символи, които не са специални за HTML.

За да използваме безпроблемно кирилица, задаваме кодиране на страницата windows-1251 чрез директивата:

```
<%@ page pageEncoding="windows-1251" %>
```

Страницата **header.jsp** първо указва на клиентския Web-браузър да не кешира текущата страница. Това е изключително важно и ако се пропусне, могат да се получат много досадни проблеми. Ако една динамична страница, която показва например всички съобщения от форума, бъде кеширана от браузъра, потребителят няма да вижда промените на съобщенията и за него форумът ще изглежда мъртъв. Подобна е ситуацията със всички динамични страници в едно Web-приложение.

След забраната на кеша на браузъра страницата **header.jsp** създава HTML таблица, слага в нея един ред, съдържащ заглавието на приложението и проверява дали в сесията има зададено съобщение за грешка, което трябва да се покаже на потребителя. Ако такова съобщение има (ако атрибутът с име **IConstants.LAST_ERROR** има някаква стойност), съобщението се премахва от сесията и се отпечатва с червени букви в отделен ред от таблицата.

Страницата **header.jsp** не е предназначена да се извиква от потребителя и затова не генерира цялостен HTML документ, а само част от документ, предназначен да бъде вграден друг документ.

Да разгледаме и завършващата част, която се добавя от Web-приложението в края на всяка JSP страница, непосредствено преди тага за затваряне на тялото на HTML документа:

footer.jsp

```
<%@ page import="miniforum.IConstants,
                 miniforum.HtmlUtils,
                 miniforum.data.UserUtils"%>
<%@ page pageEncoding="windows-1251" %>
<table border="0" width="100%">
    <tr><td align="center" bgcolor="#66CCFF">
<%
    String currentUser = UserUtils.getCurrentUser(session);
    if (currentUser != null) {
%>
        Потребител: <%=HtmlUtils.htmlEscape(currentUser)%>
           
        <a href="<%=IConstants.LOGOUT_URL%>">[изход]</a>
<%
    } else {
%>
         
<%
    }
%>
    </td></tr>
</table>
```

Както се вижда от кода, страницата генерира една HTML таблица, в която отпечатва името на текущия потребител, следвано от hyperlink за излизане от системата (връзка към действието „**doLogout**", което съответства на сървлета **LogoutServlet** от подсистемата за автентикация и контрол на достъпа). Ако няма текущ потребител, в таблицата се отпечатва единичен интервал, защото ако се остави празна, браузърът няма да я визуализира изобщо.

Съща, както и **header.jsp** тази страница не е предназначена да се извиква самостоятелно и трябва да се изпо9лзва само за вграждане в други страници.

Отново използвахме класа **miniforum.HtmlUtils** с неговия метод **htmlEscape(…)** за да избегнем появата на непозволени за HTML символи в тялото на таблицата, защото те могат да повредят документа и носят със себе си заплаха за сигурността. Сорс-кодът на класа е същият този код, който дадохме като пример за разрешаване на проблема с HTML ескейпването в една от предходните теми. Ето как изглежда той:

HtmlUtils.java

```
package miniforum;
```

```
public class HtmlUtils {
    /**
     * Escapes given text for placing it in the HTML body. If
     * you need escaping for placing text in an attribut value,
     * you should remove the escaping for the "\n" character.
     *
     * (c) Svetlin Nakov, 2004 - http://www.nakov.com
     */
    public static String htmlEscape(String aText) {
        if (aText == null) {
            return "";
        }
        StringBuffer escapedText = new StringBuffer();
        for (int i=0; i<aText.length(); i++) {
            char ch = aText.charAt(i);
            if (ch == '\'')
                escapedText.append("&#39;");
            else if (ch == '\"')
                escapedText.append("&#34;");
            else if (ch == '<')
                escapedText.append("&lt;");
            else if (ch == '>')
                escapedText.append("&gt;");
            else if (ch == '&')
                escapedText.append("&amp;");
            else if (ch == '\n')
                escapedText.append("<br>\n");
            else if (ch == ' ')
                escapedText.append(" ");
            else if (ch == '\t')
                escapedText.append("    ");
            else
                escapedText.append(ch);
        }
        String result = escapedText.toString();
        return result;
    }
}
```

Класът съдържа един единствен статичен метод за ескейпване на специалните за HTML символи и работи по много простичък начин – преминава през всички символи от подадения текст и замества тези, които са специални за езика HTML с еквивалентни на тях последователности от разрешение символи.

Нека сега разгледаме формата за влизане в системата:

login.jsp

```
<%@ page import="miniforum.IConstants"%>
<%@ page contentType="text/html;charset=windows-1251" %>
```

```
<html>
<head><title>Mini Forum - Login</title></head>
<body>
    <%@ include file="header.jsp" %>
    <br>
    <div align="center">
    <form method="post" action="<%=IConstants.LOGIN_URL%>">
        <input type="text" name="<%=IConstants.USER_PARAM%>">
        <br>
        <input type="password"
            name="<%=IConstants.PASSWORD_PARAM%>">
        <br>
        <input type="submit" value="Влез">
    </form>
    </div>
    <%@ include file="footer.jsp" %>
</body>
</html>
```

Както виждаме, в нея няма нищо сложно – съвсем обикновена HTML форма, която е поставена в съвсем обикновена HTML страница, като преди формата е включена заглавната част **header.jsp**, а след формата е включена завършващата част **footer.jsp**.

За да няма проблеми с кирилицата, формата задава за content-type стойност „**text/html;charset=windows-1251**”.

За имената на полетата от формата се използват константи, за да е по-лесно да се извлекат след това същите полета от подсистемата за автентикация и контрол на достъпа.

При submit на формата данните се изпращат към действието “**doLogin**”, което съответства на сървлета **LoginServlet**, който вече разгледахме.

Да видим сега как работи основната форма на приложението:

main.jsp

```
<%@ page import="java.util.Vector,
                 miniforum.HtmlUtils"%>
<%@ page contentType="text/html;charset=windows-1251" %>
<html>
<head><title>Mini Forum - Messages</title></head>
<body>
    <%@ include file="header.jsp" %>
    <div align="center">
        <%-- Display all messages --%>
        <%@ include file="showMessages.jsp" %>
        <br>
        <%-- Display add new message form --%>
        <%@ include file="addNewMessageForm.jsp" %>
    </div>
```

```
    <%@ include file="footer.jsp" %>
</body>
</html>
```

Макар и очакванията да са, че тази форма ще е най-сложната, това съвсем не е така. Това се дължи на доброто структуриране на отделните части от потребителски интерфейс. С цел опростяване на кода, подобряване на четимостта му и намаляване на усилията за поддръжката му формата е разбита на 4 по-прости JSP страници, които се извикват една след друга.

Първоначално се извиква заглавната част **header.jsp**, след нея се извиква страницата за показване на съобщенията от форума **showMessages.jsp**, след тях се показва формата за добавяне на ново съобщение **addNewMessageForms.jsp** и накрая се показва завършващата част **footer.jsp**. Всичко е просто и ясно. Кодът няма нужда от коментари. Той се самоописва (**self-documenting code**). Това е препоръчителният начин за изграждане на сложни приложения – чрез разбиването им на множество по-прости части.

Формата задава кодиране за отговора на HTTP заявката – windows-1251. Ако това се пропусне, е възможно кирилицата да не излезе правилно.

Да разгледаме сега JSP страницата за показване на съобщенията от форума:

showMessages.jsp

```
<%@ page import="miniforum.data.Message,
                 miniforum.HtmlUtils,
                 java.util.Vector,
                 miniforum.data.MessageUtils,
                 java.util.Enumeration"%>
<%@ page pageEncoding="windows-1251" %>
<%
    Message[] messages =
        MessageUtils.getForumMessages(application);
    if (messages.length == 0) {
%>
    <br>
    <b>Няма съобщения.</b>
<%
    } else {
        for (int i=0; i<messages.length; i++) {
            Message msg = messages[i];
            long id = msg.getID();
            String author = msg.getUser();
            String subject = msg.getSubject();
            String contents = msg.getContents();
```

```
%>
    <br>
    <table border="1" width="90%" cellspacing="0"
    cellpadding="5">
        <tr>
            <td bgcolor="lightblue">
                <i>Автор:</i> <%=HtmlUtils.htmlEscape(author)%>
                   |   
                <i>Тема:</i> <%=HtmlUtils.htmlEscape(subject)%>
                   |   
                <a href="doDelete?id=<%=id%>">
                    [изтрий]
                </a>
            </td>
        </tr>
        <tr>
            <td colspan="3">
                <font face="monospace">
                    <%=HtmlUtils.htmlEscape(contents)%>
                </font>
            </td>
        </tr>
    </table>
<%
        }
    }
%>
```

Тя е малко по-сложна от главната форма на приложението. При изпълнение тя взима от контекста на приложението съобщенията от форума, прави цикъл по тях и ги визуализира. От съображения за по-красив външен вид всяко съобщение се отпечатва в отделна HTML таблица. Ако няма нито едно съобщение, се отпечатва текст, който информира потребителя, че форумът е празен.

При визуализацията на всяко съобщение се ескейпват всички непозволени за езика HTML символи. Това става с добре познатия ни метод **`htmlEscape(…)`** на класа **`miniforum.HtmlUtils`**.

Към всяко съобщение се добавя и hyperlink за изтриване. Той се формира от действието "**`doDelete`**", като към него се добавя като параметър в URL адреса уникалния номер на съобщението. По принцип при такова динамично сглобяване на URL трябва да се използва URL encode ескейпване, но в случая това не се налага, защото номерата на съобщенията могат да бъдат само цели числа и не могат да съдържат непозволени символи. Действието "**`doDelete`**" отговаря на сървлета за изтриване **`DeleteServlet`**, който ще разгледаме след малко.

Понеже форумът позволява много потребители да работят едновременно, е възможно по време на визуализацията на съобщенията от форума да се окаже, че някои от тези съобщения са били вече изтрити малко преди това от друг потребител. Това не е проблем, защото методът, който връща всички съобщения от подсистемата за съхранение и управление на данните връща копие от тях и реално това, че някои съобщения са били преждевременно изтрити не указва влияние на визуализацията.

Като част от основната форма на приложението **main.jsp** се включва и още една страница – формата за добавяне на ново съобщение:

addNewMessageForm.jsp

```
<%@ page import="miniforum.IConstants"%>
<%@ page pageEncoding="windows-1251" %>
<form method="POST" action="doAddMessage">
    <table border="1" cellspacing="0" cellpadding="5"><tr><td>
        <table border="0">
        <tr>
            <td align="center">
                Тема: <input type="text" size="46"
                name="<%=IConstants.SUBJECT_PARAM%>">
            </td>
        </tr>
        <tr>
            <td>Съобщение:</td>
        </tr>
        <tr>
            <td>
                <textarea name="<%=IConstants.CONTENTS_PARAM%>"
                type="text" cols="40" rows="5"></textarea>
            </td>
        </tr>
        <tr>
            <td align="center">
                <input type="submit" value="Добави">
            </td>
        </tr>
        </table>
    </td></tr></table>
</form>
```

Няма нищо сложно в тази страница – тя представлява съвсем обикновена HTML форма, която при submit се изпраща към действието **doAddMessage**, което съответства на сървлета **AddMessageServlet**, който ще разгледаме след малко. Отново имената на полетата от формата са зададени с константи, за да могат по-лесно да се извлекат изпратените данни от сървлета, който ги обработва.

Подсистема за изпълнение на действията на потребителя – имплементация

Ще имплементираме подсистемата за изпълнение на действията на потребителя като съвкупност от сървлети, разположени в пакета **miniforum.action**. Тези сървлети имат грижата да обработват клиентските заявки за добавяне и изтриване на съобщения.

Ето как изглежда сорс-кода на сървлета за добавяне на ново съобщение:

AddMessageServlet.java

```
package miniforum.action;

import miniforum.data.*;
import miniforum.IConstants;

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

public class AddMessageServlet extends HttpServlet {
    protected void doPost(HttpServletRequest aRequest,
            HttpServletResponse aResponse)
            throws ServletException, IOException {
        aRequest.setCharacterEncoding("cp1251");
        HttpSession session = aRequest.getSession();
        ServletContext app = session.getServletContext();
        String currentUser = UserUtils.getCurrentUser(session);
        Message msg = new Message();
        msg.setUser(currentUser);
        String subject = aRequest.getParameter(
            IConstants.SUBJECT_PARAM);
        msg.setSubject(subject);
        String contents = aRequest.getParameter(
            IConstants.CONTENTS_PARAM);
        msg.setContents(contents);
        try {
            MessageUtils.addForumMessage(app, msg);
        } catch (IllegalArgumentException iae) {
            session.setAttribute(IConstants.LAST_ERROR,
                "Невалидно заглавие/съдържание на съобщение!");
        }
        aResponse.sendRedirect("main.jsp");
    }
}
```

Този сървлет е предназначен да бъде извикван по метод HTTP GET в резултат от изпращане на формата за добавяне на ново съобщение и очаква да получи два параметъра – заглавие и текст на новото съобщение. Всичко, което прави сървлетът е да извлече двата

параметъра, да създаде ново съобщение и да се опита да го добави във форума. Ако не се получи изключение, потребителския браузър се пренасочва към главната страница на приложението. Ако се получи изключение при добавянето, това най-вероятно означава, че съобщението е невалидно (няма заглавие или съдържание). В този случай в полето за грешка в сесията се добавя подходящо съобщение за грешка, което да бъде показано на потребителя и се извършва пренасочване към главната страница на приложението.

Да разгледаме сървлета за изтриване на съобщения от форума:

DeleteServlet.java

```
package miniforum.action;

import miniforum.data.MessageUtils;
import miniforum.IConstants;

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

public class DeleteServlet extends HttpServlet {
    protected void doGet(HttpServletRequest aRequest,
            HttpServletResponse aResponse)
            throws ServletException, IOException {
        HttpSession session = aRequest.getSession();
        String idParameter = aRequest.getParameter(
            IConstants.ID_PARAM);
        try {
            int id = Integer.parseInt(idParameter);
            ServletContext application =
                session.getServletContext();
            MessageUtils.deleteForumMessage(id, application);
        } catch (Exception ex) {
            session.setAttribute(IConstants.LAST_ERROR,
                "Не мога да изтрия съобщението!");
        }
        aResponse.sendRedirect(IConstants.MAIN_FORM);
    }
}
```

Сървлетът е предназначен да бъде извикван от hyperlink по метод HTTP GET от главната форма на приложението. Целта на сървлета е да изтрива съобщение от форума по зададен уникален идентификационен номер. Всичко, което прави сървлетът е да извлече параметъра, указващ номера, съобщението, което трябва да бъде изтрито и да се опита да го изтрие. При успех потребителският браузър се пренасочва към главната форма на приложението, а при неуспех в сесията се поставя съобщение

за грешка и също се извършва пренасочване към главната форма. Причините за неуспех при изтриването може да са няколко – от липса на стойност за параметъра, който указва номера на съобщението за изтриване, до провал на изтриването заради това, че друг потребител го е изтрил преждевременно.

Deployment на примерното приложение „Мини форум”

Вече имаме сорс-кода на цялото приложение. За да го накараме да заработи, трябва да създадем конфигурационния файл **web.xml**, в който да опишем сървлетите и сървлет филтъра и да разположим всички файлове на определените за тях места съгласно спецификацията за Web-приложения на J2EE.

Да започнем с конфигурационния файл на Web-приложението:

web.xml

```
<?xml version="1.0"?>

<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <filter>
        <filter-name>AuthenticationFilter</filter-name>
        <filter-class>
            miniforum.action.AuthenticationFilter
        </filter-class>
    </filter>

    <filter-mapping>
      <filter-name>AuthenticationFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>
            miniforum.action.LoginServlet
        </servlet-class>
    </servlet>

    <servlet>
        <servlet-name>AddMessageServlet</servlet-name>
        <servlet-class>
            miniforum.action.AddMessageServlet
        </servlet-class>
    </servlet>
```

```
    <servlet>
        <servlet-name>DeleteServlet</servlet-name>
        <servlet-class>
            miniforum.action.DeleteServlet
        </servlet-class>
    </servlet>

    <servlet>
        <servlet-name>LogoutServlet</servlet-name>
        <servlet-class>
            miniforum.action.LogoutServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/doLogin</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>AddMessageServlet</servlet-name>
        <url-pattern>/doAddMessage</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>DeleteServlet</servlet-name>
        <url-pattern>/doDelete</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>LogoutServlet</servlet-name>
        <url-pattern>/doLogout</url-pattern>
    </servlet-mapping>
</web-app>
```

Файлът започва с индикацията, че това е XML документ с версия 1.0 на XML стандарта, без да е зададено кодиране, което означава, че се подразбира кодирането “UTF-8”.

Следва дефиницията, която указва, че този документ отговаря на DTD (Document Type Definition) структурата описана от спецификацията за Java-базирани Web-приложения “Servlets 2.3”. Тази DTD дефиниция (http://java.sun.com/dtd/web-app_2_3.dtd) описва позволените тагове във файла **`web.xml`**, както и последователността, в която могат да се срещат.

Следва описанието на филтъра за автентикация. Във файла е указано, че всички ресурси от приложението задължително трябва да се обработят от филтъра **`miniforum.action.AuthenticationFilter`** преди да бъде направен опит за достъп до тях.

Следват описанията на сървлетите за влизане в системата, добавяне на съобщение, изтриване на съобщение и излизане от системата. Във файла е описано следното съпоставяне на относителни URL адреси към сървлетите от приложението:

относителен URL	сървлет клас
/doLogin	**miniforum.action.LoginServlet**
/doAddMessage	**miniforum.action.AddMessageServlet**
/doDelete	**miniforum.action.DeleteServlet**
/doLogout	**miniforum.action.LogoutServlet**

След като вече имаме всички файлове, съставящи приложението, трябва да компилираме всичкия Java сорс-код (**.java** файловете) и да получим от тях съответните **.class** файлове. След това трябва да създадем директория с име **mini-forum**, да създаден в нея директориите **WEB-INF** и **WEB-INF\classes** и да разположим файловете на приложението по следния начин:

```
WEB-INF/
    classes/
        miniforum/
            action/
                AddMessageServlet.class
                AuthenticationFilter.class
                DeleteServlet.class
                LoginServlet.class
                LogoutServlet.class
            data/
                Message.class
                MessageUtils.class
                UserUtils.class
            HtmlUtils.class
            IConstants.class
    web.xml
addNewMessageForm.jsp
footer.jsp
header.jsp
login.jsp
main.jsp
showMessages.jsp
```

Цялата директория **mini-forum**, заедно със всичките файлове в нея, разположени, както е зададено на схемата, образуват нашето Web-приложение, което най-сетне е готово за стартиране.

По желание можем да съберем всичко в един **.war** файл като запишем съдържанието на цялата директория в ZIP архив с име **forum.war**. Можем да направим това или с **WinZip**, **gzip** или друга компресираща

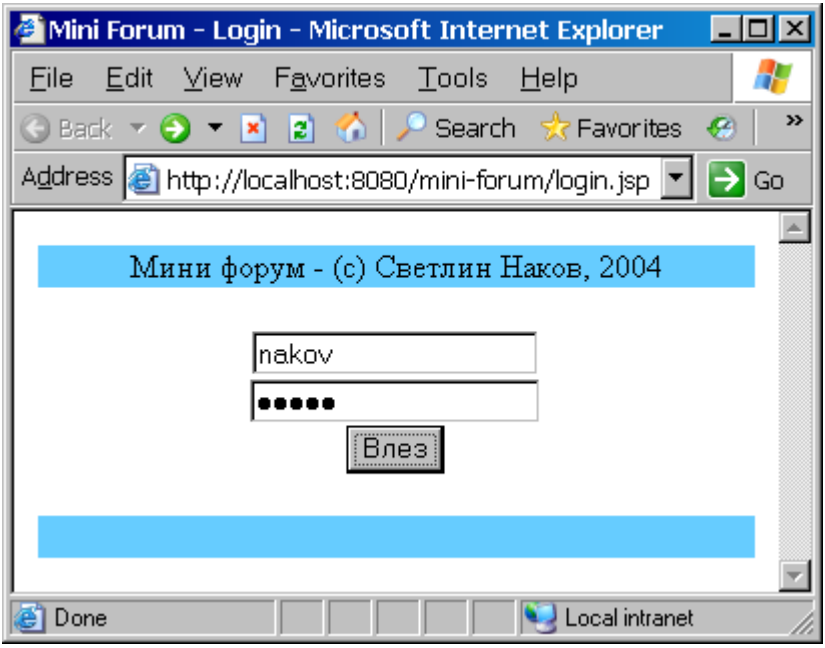
програма, или чрез следната команда, изпълнена от конзолата в директорията на приложението:

```
jar -cf ..\mini-forum.war *.*
```

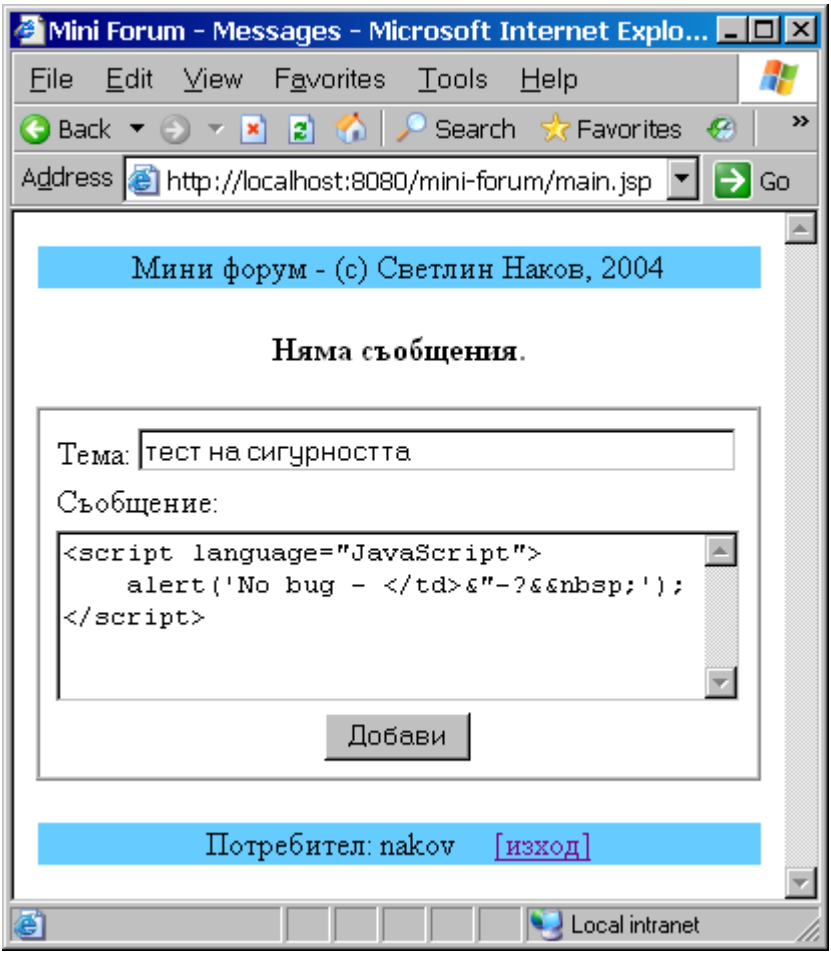
Полученият архив **mini-forum.war** може да се използва за deploy-ване на Web-приложението на всеки стандартен Web-контейнер.

Примерното приложение „Мини форум" в действие

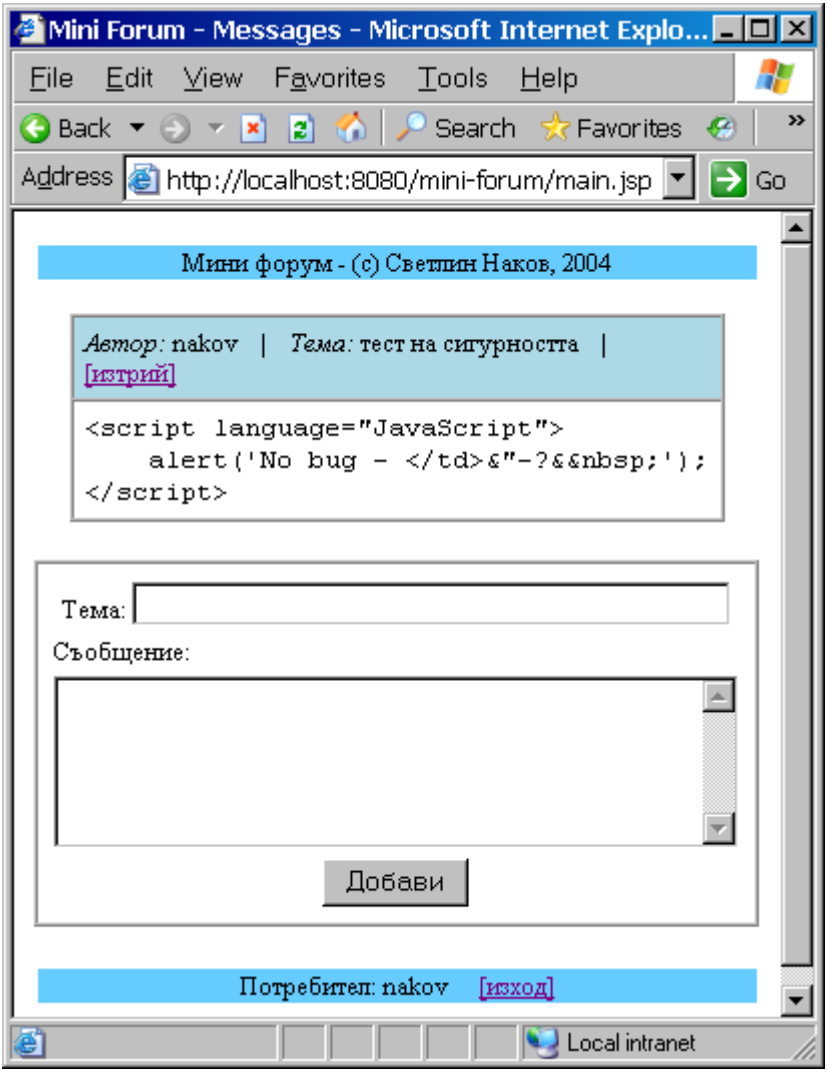
След като копираме директорията на приложението в поддиректория **webapps** на Web-контейнера Tomcat 5.0 и стартираме Tomcat, можем да видим резултата от изпълнението му от всеки стандартен Web-браузър. Ето как изглежда формата за влизане във форума, когато в системата няма влязъл потребител:



След успешна автентикация с потребителското име **nakov** и тайната парола **nakov** системата ни препраща към основната форма:



След успешно добавяне на ново съобщение, се получава основната форма на приложението добива следния вид:



Имаме възможност да добавяме, да изтриваме съобщения, да разглеждаме съобщенията и да излезем от системата – всичко, което си бяхме поставили като задача.

НАПС
НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» ***Светлин Наков*** е преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

Той е автор на десетки научни и технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

През 2004 г. получава наградата "**Джон Атанасов**" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество.

» ***Мартин Кулов*** е изпълнителен директор във фирма "Код Атест", където разработва проекти за повишаване качеството на софтуерните продукти в България чрез автоматизация на процесите и внедряване на системи за управление на качеството.

Мартин е опитен лектор и сертифициран от Майкрософт разработчик по програмите MCSD и MCSD.NET.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда задълбочени курсове по разработка на софтуер и съвременни софтуерни технологии.

» Предлагани специалности:

.NET Enterprise Developer

Java Enterprise Developer

» **Качествено обучение** с много **практически упражнения**

» Завършвате само за 3 месеца.

» **Гарантирана работа** след успешно завършване!

» Професионална сертификация!

» **БЕЗПЛАТНО!**

Учите безплатно, плащате като завършите и започнете работа.

Стипендии от софтуерни фирми.

http://academy.devbg.org

Поглед към следващото издание на книгата

В една книга винаги има нещо, което може да се добави, винаги има нещо, което може да се подобри и винаги има нещо, което може да се обясни по-добре. Така е и с тази книга – има много възможности за подобрение, но трябва да почакаме да излезе следващата версия.

В следващото издание наред с изчистването на грешките, ще се постараем да добавим и следните допълнителни теми, за които така и не остана време:

- към първа глава:
  - работа с асинхронни сокети (non-blocking sockets);
  - достъп до защитени ресурси през HTTPS;
  - работа с електронна поща (SMTP, POP3, JavaMail API)
- към втора глава:
  - Java аплетите и Swing;
  - разработка на chat аплет;
- към трета глава:
  - прихващане на събитията от жизнения цикъл на Web-приложението (стартиране, спиране, създаване на сесия и изтичане на сесия);
  - потребителски тагове и библиотеки от тагове (custom tags and tag libraries);
  - библиотеката JSTL (Java Server Pages Standard Tag Library);
  - framework за Web-приложения Struts.

Надяваме се, че дори и без тези теми книгата е полезна и дава една добра основа за развитие в областта на програмирането за Интернет със средствата на Java платформата.

Заключение

Силно се надявам, ако с тази книга не съм успял да ви науча на нещо ново и полезно за Интернет програмирането със средствата на езика Java, то поне да съм успял да ви объркам на едно по-високо, професионално ниво.

Светлин Наков
април, 2004

Написана от програмист с дългогодишен опит и уважаван преподавател по съвременни софтуерни технологии, тази книга е ценно ръководство за всички, които искат да навлязат в света на програмирането за Интернет.

Въпреки, че е ориентирана към Java разработчици, книгата учи на основни концепции, независими от езика за програмиране – многонишково програмиране, синхронизация, класически синхронизационни проблеми, устройство и организация на Интернет, принципи на комуникацията по протоколи TCP/IP, принципи на Web-програмирането, технологии за динамично генериране на Web-съдържание, протоколи и стандарти.

Авторът се е спрял на езика Java не само заради лесната, с която се използва, голямата му популярност и широкото му разпространение, но и заради силно развитите му средства за разработка на Интернет-ориентирани приложения.

За да навлезете в Интернет-ориентираното програмиране с Java не е необходимо да сте програмирали на този език. Достатъчно е да имате начални знания по обектно-ориентирано програмиране и да познавате синтаксиса на Java. Всичко останало, което ви трябва, ще го научите от книгата. Ще се запознаете с принципите на сокет-базираната комуникация, с технологията на Java аплетите и със средствата на Java платформата за Web-програмиране – Java сървлети, JSP страници и Web-приложения.

Учебният материал от книгата вече няколко години се използва успешно в курса „Интернет програмиране с Java" в Софийски Университет „Св. Климент Охридски", където авторът заедно със свои колеги е водил лекции по него на повече от 700 студента.

Информацията е поднесена в изключително разбираем стил и с изтъкнато майсторство – последователно, структурирано, изчерпателно и с много, много примери. Всяко нещо е илюстрирано с подходящ, внимателно подбран пример – най-ценното в една книга за програмиране.

доц. д-р Божидар Сендов
Софийски Университет „Св. Климент Охридски"